AF ZJW

PATENT

ATTORNEY DOCKET NO. 114596-28-000053BS

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Serial No.:	09/626,325	Confirmation No.:	7939
Applicant:	John S. Yates, Jr., et al.		
Title:	OPERATING SYSTEM FOR COMPUTER WITH TWO ARCHITECTURES		
Filed:	July 26, 2000	Art Unit:	2183
Atty. Docket:	114596-28-000053BS	Examiner:	R. Ellis

CERTIFICATE OF MAILING (37 C.F.R. § 1.8a)

Art Unit 2183, SPRE Shop
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

I hereby certify that the attached

- Return postcard
- This Certificate of Mailing
- Petition for Review by Technology Center SPRE (with Exhibits A-K)

(along with any paper(s) referred to as being attached or enclosed) are being deposited with the United States Postal Service on the date shown below with sufficient postage as first-class mail in an envelope addressed to Art Unit 2183, SPRE Shop, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

Respectfully submitted,

WILLKIE FARR & GALLAGHER LLP

Dated: September 15, 2005

By: _____
David E. Boundy
Registration No. 36,461

WILLKIE FARR & GALLAGHER LLP
787 Seventh Ave.
New York, New York 10019
(212) 728-8757
(212) 728-9757 Fax

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Serial No.:	09/626,325	Confirmation No.:	7939
Applicant:	John S. Yates, Jr., et al.		
Title:	OPERATING SYSTEM FOR COMPUTER WITH TWO ARCHITECTURES		
Filed:	July 26, 2000		
Art Unit:	2183		
Examiner:	R. Ellis		
Atty. Docket:	114596-28-000053BS		
Customer No.	38492		

AFTER FINAL – EXPEDITED PROCEDURE

PETITION FOR REVIEW BY TECHNOLOGY CENTER SPRE

Art Unit 2183, SPRE Shop
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Petitioner requests review by the Director's representative in the Technology Center of the following two issues that arise out of the Office Action of July, 19, 2005:

1. Did the examiner err in objecting to the "Amendment Accompanying RCE" of April 27, 2005 as "new matter?"

Yes. An amendment need not have literal support in the specification, as the examiner requires; "inherent" support is entirely sufficient. Here, the amendment to the specification merely states the "ordinary meaning" in the art of a term, and the ordinary theory of operation, and thus is not new matter.

I certify that this correspondence, along with any documents referred to therein, is being transmitted by facsimile, without exhibits, on September 15, 2005 to Art Unit 2183 at FAX no. 571 273 8300, and deposited with the United States Postal Service, with exhibits, on September 15, 2005 as First Class Mail in an envelope with sufficient postage addressed to Art Unit 2183, SPRE Shop, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

2. May this first Office Action after an RCE be made final?

No. Examination is incomplete – that which is incomplete cannot be final. An Information Disclosure Statement that was timely filed and cited in the Request for Continued Examination has not been considered. The examiner states that he disregards the effect of the amendment because of the "new matter" issue; the MPEP instructs otherwise. Until the application has been timely and completely examined in the manner required by the Director, final rejection is premature.

This Petition is timely presented within two months of the examiner's action of July 19, 2005.

I. The Amendment of April 27, 2005 Adds No "New Matter"

The Office Action of July 19, 2005 "objects" to an amendment to the specification proposed in the "Amendment Accompanying RCE" of April 27, 2005. There is no "new matter" or § 112 ¶ 1 rejection of any claim. Thus, MPEP § 608.04(c) provides that the "objection" is reviewable by petition.

A. The Text of the Amendment

The amendment to which the examiner "objects" is the underlined portion of the following. As will be shown below, the underlined language falls into three categories recognized not to be "new matter:" (a) legal principles of interpretation that are inherent in every patent application, and (b) the ordinary and customary meanings of several terms, as those terms are understood and actually used by those of skill in the relevant art, inherent in the use of the terms themselves, and (c) the ordinary theory of operation of "processes" and "thread," which is also implicit in the terms themselves:

> Referring to **Figs. 3a** and **3l** and to Table 1, X86 threads (*e.g.*, **302, 304**) managed by X86 operating system **306**, carry the normal X86 context, including the X86 registers, as represented in the low-order halves of r32-r55, the EFLAGS bits that affect execution of X86 instructions, the current segment registers, etc. (The terms "process" and "thread" are used herein in their ordinary and customary, though formal, senses, as actually used in the programming language systems, operating systems, and processor architecture arts. Generally, a "process" is a unit of processor scheduling and protection, each with an associated data structure (or set of data structures) that, in most implementations, holds machine register values and other context associated with the process. The process data structures, and thus the processes of a computer, are usually under the management of an operating system, usually the operating system's scheduler. Generally, a "thread" is a flow of control within a process. Each thread has an associated data structure (or set of data structures)

> that, in most implementations, hold machine register values (usually different than the registers associated with a process) and other context associated with the thread. The thread data structures, and thus the threads of a process, are usually managed either by an operating system or other run time system, to permit the thread to be scheduled independently of and concurrently with other threads of the same process.) In addition, if an X86 thread **302**, **304** calls native Tapestry libraries **308**, X86 thread **302**, **304** may embody a good deal of extended context, the portion of the Tapestry processor context beyond the content of the X86 architecture. A thread's extended context may include the various Tapestry processor registers, general registers r1-r31 and r56-r63, and the high-order halves of r32-r55 (see Table 1), the current value of ISA bit **194** (and in the embodiment of section IV, *infra*, the current value of XP / calling convention bit **196** and semantic context field **206**).

This amendment was made to resolve a dispute relating to the term "thread." For nearly two years, Applicant has encouraged the examiner to interpret the term "thread" in its ordinary and customary, though formal, sense, as actually used in the arts relevant to this application: programming language systems, operating systems, and processor architecture. As noted below, the precise definition of "thread" varies somewhat vendor-to-vendor, but is consistent in one respect: every relevant dictionary and technical reference that states a formal definition of the term "thread" emphasizes that "threads" provide one or more concurrent or independent flows of control.[1]

In contrast, the examiner relies on an extrapolation of his own devising, a definition that is not found in any dictionary or any other source, and for which he has been unable to provide a single example of actual use in the art. The underlying sources from which he extrapolates are obsolete, and provide only irrelevant general definitions, rather than the definition that is specific to the relevant arts. The examiner defines "thread" as any "process that is part of a larger process or program," including anything that "handles an interrupt" (Office Action of 10/27/2004, page 2, last 3 lines), even if the handler is structured specifically to prevent concurrent execution.

---

[1] As will be shown below, the art allows for single-thread processes, in the simplest case. "Concurrent" in the relevant arts "[pertains] to the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution. *Synonym*: parallel. *Contrast*: simultaneous." Authoritative Dictionary of IEEE Standards Terms, 7th ed. (2000).

B. Every Definition of "Thread" In the Relevant Art Implies "Concurrency"

1. Vendor-Neutral Definitions of "Thread" and "Process" Uniformly Note that Threads Allow "Concurrency"

Various technical dictionaries give the following definitions of the terms "process" and "thread" in the specific context of the arts relevant to this application. Note that every single definition in the specific arts of this application is consistent in stating that "concurrency" (or "parallelism" or "independent scheduling") is the basic "reason for being" of threads.

The Authoritative Dictionary of IEEE Standards Terms, 7th ed. (2000) (Exhibit A) collects formal definitions from various formal standards issued by various official and non-governmental organization standards bodies. The IEEE Dictionary gives the following relevant definitions (underline added):

> **thread (4)** A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, *errno* value, thread-specific key/value bindings, and the required system resources to support a flow of control. ...[2]
>
> **thread of control** A sequence of instructions executed by a conceptual sequential subprogram, independent of any programming language. More than one thread of control may execute concurrently, interleaved on a single processor, or on separate processors. The conceptual threads of control in an Ada application are Ada tasks. They may, but need not, correspond to the POSIX threads defined in POSIX.1.[3]

In definition "thread (4)," the idea of "concurrency" is inherent in the terms "scheduling priority and policy:" scheduling which of several threads is to run at any given time only makes sense if they can execute concurrently.

The examiner himself (Office Action of 10/27/2004, page 2) cited the following definition as his first choice definition, in the Office Action of 10/27/04. The examiner found this definition on the Yahoo! web site, and Yahoo attributes the definition to The American

---

[2] Definition (4) originated with the International Organization for Standardization and International Electrotechnical Commission, in their ISO/IEC Standard No. 9945-1-1996, Portable Operating System Interface (POSIX), the formal definition for UNIX operating systems.

The IEEE Dictionary gives another definition, "thread (3) A single sequential flow of control within a process." However, definition (3) is stated only in standards that relate to conformance testing of software developed to implement definition (4). Thus, concurrency is inherently part of definition (3) as well.

[3] The POSIX definition is definition (4).

Heritage Dictionary of the English Language: Fourth Edition (2000) (Exhibit B) (underline added):

> **thread**, n. *Computer Science* a. A portion of a program that can run independently of and concurrently with other portions of the program.

The "Wikipedia" web site is generally regarded as an accurate resource. Wikipedia defines "thread" as follows http://en.wikipedia.org/wiki/Thread_%28computer_science%29 (Exhibit C) (bold in original, underline added)[4]

> Many programming languages, operating systems, and other software development environments support what are called "**threads**" of execution. Threads are similar to processes, in that both represent a single sequence of instructions executed in parallel with other sequences, either by time slicing or multiprocessing. ...
>
> An advantage of a multi-threaded program is that it can operate faster on computer systems that have multiple CPUs, or across a cluster of machines. This is because the threads of the program naturally lend themselves for truly concurrent execution....
>
> Threads allow a program to do multiple things concurrently. ...
>
> **Implementations**
>
> There are many different and incompatible implementations of threading. These can either be kernel-level or user-level implementations. ...

Within months of the filing date of this application, the specialized definition of the term "thread" had become integrated into **freshman-level** computer science courses. For example, slides for a Fall 2000 freshman-level course (http://www.cse.ucsd.edu/classes/fa00/cse120/lectures/5-threads.pdf, Exhibit D) at the University of California, San Diego explains threads as follows (underline added):

- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities

---

[4] A printed copy of this definition from http://www.wordiq.com/definition/Thread_%28computer_science%29 was included with Applicant's paper of July 26, 2004.

Lecture notes from January 2000, http://www.andrew.cmu.edu/course/15-412/ln/412springlecture4.html (Exhibit E), describe the relationship of processes and threads that was taught at Carnegie-Mellon University. "Context switch" is another term for scheduling, for sharing execution among several threads or processes that are all ready to execute concurrently (italic and bold in original, underline added):

> **Threads, a.k.a Light-Weight Processes (LWP)**
>
> Now suppose that we want multiple process-like things that are separately schedulable -- but share memory. ...
>
> Now consider multiple "processes" sharing the same memory, but otherwise maintaining different state (registers, &c). We call these processes within processes *threads*. We call them this, because they are separate *threads of control* within the same process....
>
> - Fast context switch: ...
>
> ...
>
> The PCB now maintains the state of each thread and allows each thread to be scheduled independently. ...

2. Vendor-Specific Definitions of "Thread" Uniformly State a Requirement for "Concurrency"

By 1999, many different operating systems implemented some form of "threads." Definitions drawn from various manufacturers' documentation vary somewhat in detail, but are consistent in their emphasis on "concurrency," "parallelism," or "independent scheduling."

At its Microsoft Developer's Network (Microsoft's technical library directed to those who must have precise and accurate information, as opposed to "home users," the audience for the Microsoft Computer Dictionary), Microsoft describes "threads" as follows http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/about_processes_and_threads.asp, multiple_threads.asp and creating_threads.asp (Exhibit F) (italic and bold in original, underline added):

> **About Processes and Threads**
>
> A *thread* is the entity within a process that can be scheduled for execution. ...
>
> Microsoft® Windows® supports *preemptive multitasking*, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

Creating Threads

The CreateThread function creates a new thread for a process. ... A process can have multiple threads simultaneously executing the same function.

Hewlett-Packard describes its implementation of threads in http://docs.hp.com/en/B2355-90695/pthread.3T.html (Exhibit G):

THREAD OVERVIEW

A *thread* is an independent flow of control within a process, composed of a context (which includes a register set and a program counter) and a sequence of instructions to execute.

All processes consist of at least one thread. Multi-threaded processes contain several threads. All threads share the common address space allocated for the process. ...

Sun Microsystems' documentation on threads in its Solaris variant of UNIX (http://www.sun.com/software/whitepapers/solaris9/multithread.pdf, Exhibit H) reads as follows:

Introduction

Multithreading is a popular programming and execution model that allows multiple threads to exist within the context of a single process, sharing the process' resources but able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. ...

Digital Equipment Corp. (since acquired by Hewlett-Packard) implemented "DECthreads" in its VMS operating system. The January 1999 documentation on DECthreads reads as follows, http://h71000.www7.hp.com/doc/72final/6493/6101pro.html#intro_threads_chap (Exhibit I) (underline added):

1.1 Advantages of Using Threads

Multithreaded programming means organizing and coding a program so that instances of its routines, called threads, can execute concurrently in the same process. You use threads to improve a program's performance – that is, its throughput, computational speed, responsiveness, or some combination.

Using threads can improve a program's performance on uniprocessor systems by permitting the overlap of input, output, or other slow operations with computational operations. Threads are useful in driving slow devices such as disks, networks, terminals, and printers. A multithreaded program can perform other useful work while waiting for the device to produce its next event, such as the completion of a disk transfer or the receipt of a packet from the network.

Even Apple, often the exception to the rule, uses the term "thread" consistently with every other vendor http://developer.apple.com/documentation/Cocoa/Conceptual/Multithreading/index.html#//apple_ref/doc/uid/10000057i (Exhibit J) (underline added):

> **Threads**
>
> In Mac OS X, each process comprises one or more threads. A thread is a stream of execution that runs code for the process. Multiple threads may execute the same code, but they do so independently of other threads....
>
> Threads let your program perform multiple tasks in parallel. For example, you can use threads to perform several, lengthy calculations while your user interface continues to respond to user commands. You could also use threads to divide a large job into several smaller jobs.

A document by David Graves of Hewlett-Packard, http://devresource.hp.com/drc/STK/docs/refs/sol_threads.jsp (Exhibit K), compares and contrasts the major implementations of threads in several operating systems. He notes that "concurrency" is one of the properties that unites all vendors' implementations of threads (underline added):

> **Threads on HP-UX, Solaris, and NT**
>
> This document provides a conceptual mapping of thread function calls between four implementations: HP-UX threads (based on POSIX and X/Open), Solaris threads, POSIX threads on Solaris, and [Microsoft Windows] NT proprietary threads.
>
> ...
>
> **Synchronization**
>
> Since threads run concurrently and share resources, synchronization mechanisms are required to provide mutually exclusive access to shared data. ...

C. The Theory of Operation of "Threads" Was Well-Understood by the Filing Date

Not only was the definition of "thread" well established at the filing date of this application, the theory of operation was so well understood that they were part of the undergraduate curriculum. In particular, the usual implementation involved a series of data structures (sometimes called "control blocks" or "context"), managed either by the operating system or by a run-time library.

For example, Exhibit D, the **freshman-level** slides from UCSD, describes the various attributes of process and threads that must be stored in data structures on a context switch between processes or threads (underline added):

Processes

- Recall that a process includes many things
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)
- Creating a new process is costly because of all of the data structures that must be allocated and initialized ...

The Carnegie-Mellon lecture notes (Exhibit E) describe more of the theory of operation and implementation. In particular, the Carnegie-Mellon notes describe two particular data structures, the "process control block" and "thread control block" that may be used to implement processes and threads (italic and bold in original, underline added):

The Process Control Block (PCB)

We've already discussed several different types of hardware state that are associated with a process. In addition to the hardware-context, there is also the software-context of the process. This includes the state of the programs memory as well as the information that the operating systems maintains about each process. This information is stored in an operating system structure called the *process control block* (pcb). Among other things, the PCB contains the following:

- The *process ID* of the process - a unique number that identifies or names the process within the operating system
- The group ID - a number that identifies the group or classification of users to which the process belongs.
- Information about open files
- Accounting information (CPU time used, bytes read/written, &c)
- Current state (BLOCKED, READY, RUNNING) (more later)
- Linked lists and queue pointers (more later)
- Exit status that is maintained for wait (more later)

...

Threads, a.k.a Light-Weight Processes (LWP)

...

Now suppose that we want multiple process-like things that are separately schedulable -- but share memory. ...

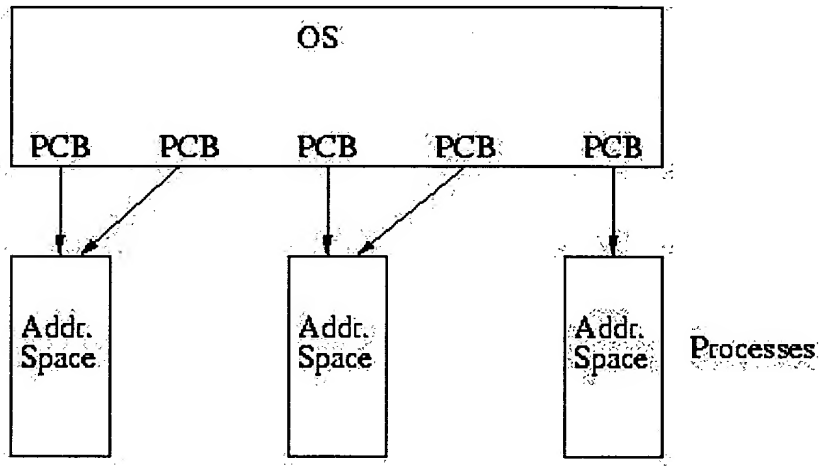
Now consider multiple "processes" sharing the same memory, but otherwise maintaining different state (registers, &c). We call these processes within processes *threads.* We call them this, because they are separate *threads of control* within the same process....

- Fast context switch: This actually depends on the implementation, but switching among threads within the same address space is faster than

switching among processes in different address spaces. We don't need to save and restore the context, including the BASE and LIMIT registers, and other memory-management registers, to context-switch. But depending on how the threads are implemented, the amount of other overhead can vary: it can be very, very fast, or just somewhat faster.

- A special cache, called the TLB doesn't need to be flushed to context switch among threads in the same address space. This not only saves the time it takes to flush the cache, but also maintains the utility of the cache -- this is a big win TLB misses are very expensive.
- A process can do useful work, even while it is blocked: yes, but this also depends on the implementation.

Kernel Supported Threads



We can think of kernel supported threads as a system where multiple PCBs can point to the same address space. Each PCB is really no longer a PCB, but more of a *thread control block (TCB)*. But it is still usually called the PCB.

The PCB now maintains the state of each thread and allows each thread to be scheduled independently. The PCB still holds the usual hardware state, queue state, pointer to address state, and a pointer to the *task control block* (TCB).

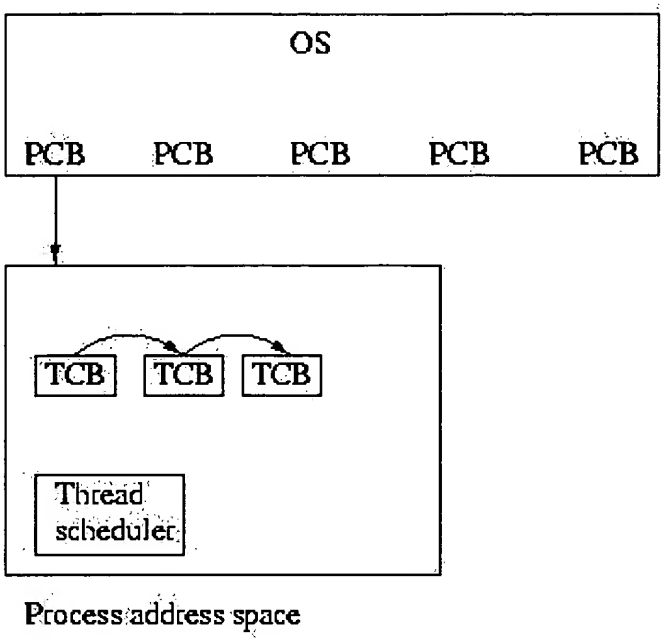
What is a TCB (notice that we are referring to a task control block, **not** a thread control block)? It maintains the state that is not stored in the PCB -- the state that applies to all threads.

...

In this case the indirection via the TCB allows us to have both common and separate state for the threads.

As the name implies, kernel supported threads require OS support. Many, but not all, OSs support kernel threads.

User-level Threads



User-level threads are implemented via a user-level thread library. The kernel neither knows nor cares that they exist. From the kernel's perspective, they are just regular code. User level threads require no changes to the kernel.

TCBs are simply malloc'd each time that a thread is created and linked together to form queues -- within the space of the process.

Context switches among the threads are very cheap -- neither the hardware nor the OS knows or cares about the context of the threads. Since we are not changing address spaces or process state, we are not changing any registers. The state of the threads is just part of the state of the process. This saves much overhead.

The Microsoft Developers' Network Articles explain some of the data structures involved (Exhibit F):

About Processes and Threads

...

A *thread* is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

The DEC/HP OpenVMS documentation (Exhibit I) describes the data structures used in the theory of operation of processes and threads:

> **6.1 Process Control Blocks (PCBs) and Process Headers (PHDs)**
>
> Two primary data structures exist in the OpenVMS executive that describe the context of a process:
>
> * Software process control block (PCB)
>
> * Process header (PHD)
>
> The PCB contains fields that identify the process to the system. The PCB comprises contexts that pertain to quotas and limits, scheduling state, privileges, AST queues, and identifiers. In general, any information that must be resident at all times is in the PCB. Therefore, the PCB is allocated from nonpaged pool.
>
> ...
>
> 6.1.1 Effect of a Multithreaded Process on the PCB and PHD
>
> With multiple execution contexts within the same process, the multiple threads of execution all share the same address space but have some independent software and hardware context. This change to a multithreaded process impacts the PCB and PHD structures and any code that references them.
>
> Before the implementation of kernel threads, the PCB contained much context that was per process. With the introduction of multiple threads of execution, much context becomes per thread. To accommodate per-thread context, a new data structure---the kernel thread block (KTB)--- is created, with the per-thread context removed from the PCB. However, the PCB continues to contain context common to all threads, such as quotas and limits. The new per-kernel thread structure contains the scheduling state, priority, and the AST queues.

D. Under the Correct Legal Test, There is No "New Matter"

It is entirely permissible for an amendment to add language to a specification, even though there is no literal support. As the courts have noted, "inherent" support is entirely sufficient. *In re Wright*, 343 F.2d 761, 767, 145 USPQ 182, 188 (CCPA 1965) ("while new language has certainly been added, we are not prone to view all new 'language' ipso facto as 'new matter.' ") The Chisum treatise sets out several classes of new language that are not new matter, Chisum, Patents § 11.04[2][a] (footnotes omitted, underline added):

> Court decisions state, in various ways, that specifications may be amended to "clarify" the original disclosure; thus: (1) "insertion by way of amendment in the description .... of a patent application do not invalidate the patent, if they are only in amplification and explanation of what was already reasonably indicated to be within the invention"; (2) amendments may be made to patent application for the purpose of curing defects, obvious to one skilled in the art, in the ... written descriptions of inventions"; (3) "an amendment to an

> application is not 'new matter' ... unless it discloses 'an invention, process or apparatus not theretofore described.' ... If the later-submitted material accused of being 'new matter' simply clarifies or completes the prior disclosure it cannot be treated as 'new matter' ..." (4) "the amendments to the specification merely render explicit what had been implicitly disclosed originally, and, while new language has certainly been added, we are not prone to view all new 'language' ipso facto as 'new matter.'"

Additionally, the consistent view has been that adding an explicit statement of the ordinary meaning of a term is not "new matter," because that ordinary meaning is inherent in the use of the term. *Ex parte Prince*, 77 USPQ 479, 481 (Pat. Off. Bd. App. 1947); *Ex parte Kharasch*, 67 USPQ 21, 22 (Pat. Off. Bd. App. 1943) (permitting addition of an explicit definition of the term "organic catalyst"); MPEP § 2163.07(I) ("The mere inclusion of dictionary or art recognized definitions ... would not be considered new matter.").

As will be shown below, every sentence of the amendment is either a statement of a legal principle that is an inherent rule of patent interpretation, a definition of a technical term drawn from the art that is inherent in the very use of the term, or an explicit statement of established theories of operation. All of these are well established as not "new matter."

The first sentence objected to is as follows:

> The terms "process" and "thread" are used herein in their ordinary and customary, though formal, senses, as actually used in the programming language systems, operating systems, and processor architecture arts.

This is merely a restatement of the legal test for "broadest reasonable meaning of the words in their ordinary usage as they would be understood by one of ordinary skill in the art." That rule of construction is inherent in every patent application. MPEP §§ 2111, 2111.01. It is not new matter. The next sentence added reads as follows:

> Generally, a "process" is a unit of processor scheduling and protection, each with an associated data structure (or set of data structures) that, in most implementations, holds machine register values and other context associated with the process.

This merely consolidates definitions (6) and (8) of "process" from the IEEE Dictionary (see page 16 of this paper), and the theory of operation for processes described in the UCSD and Carnegie-Mellon lecture notes ("address space," "Execution state (PC, SP, regs, etc.)," "data structures" and "process control block"). Because this sentence merely states the common understanding and theory in the specialized art, previously inherent, this sentence is recognized as not "new

matter." *Wright*, 343 F.2d at 767, 145 USPQ at 188; MPEP § 2163.07(a) ("The application may later be amended to recite the ... theory ... without introducing prohibited new matter.")

> The process data structures, and thus the processes of a computer, are usually under the management of an operating system, usually the operating system's scheduler.

This is merely an explicit statement consolidating the theory of operation, as reflected in the UCSD and Carnegie Mellon lecture notes (exhibits D and E), and the DEC OpenVMS description (Exhibit I) that were previously inherent. This is not new matter. The amendment continues:

> Generally, a "thread" is a flow of control within a process.

This is a direct quote of the ordinary definition of "thread" from the IEEE Dictionary (Exhibit A), and thus is not new matter. *Prince*, 77 USPQ at 481; *Kharasch*, 67 USPQ at 22.

> Each thread has an associated data structure (or set of data structures) that, in most implementations, hold machine register values (usually different than the registers associated with a process) and other context associated with the thread.

This is merely an explicit statement consolidating the definition of "thread (4)" from the IEEE Dictionary, and the theory of operation, as reflected in the UCSD slides and the Carnegie Mellon lecture notes (Exhibits D and E, see discussion "thread control block (TCB)"), and section 6.1.1 of the OpenVMS description (Exhibit I). This is not new matter.

> The thread data structures, and thus the threads of a process, are usually managed either by an operating system or other run time system, to permit the thread to be scheduled independently of and concurrently with other threads of the same process.

This is merely a statement of theory of operation from the Carnegie Mellon lecture notes (sections "Kernel Threads" and "User Level Threads"). It is not "new matter."

The addition to the specification merely reinforces the ordinary meaning in the art, and forecloses reliance on irrelevant and unreasonable definitions that are inconsistent with the specification – definitions that were impermissible before the amendment, and merely become impermissible for an additional reason now. Because the amendment induces no change to the subject matter of the specification or the claims, there is no "new matter."

Statements of ordinary meaning are given special deference when there is no reasonable alternative to the term at issue. *Kharasch*, 67 USPQ at 22 ("It seems impossible to find a term that would be of the exact scope ... It is our view that applicant is entitled ... to employ the simple broad term 'catalyst'" as supplemented by the definition added by amendment). Here, the examiner does not dispute that there is no other term in the art that has the same range of meanings as the formal sense of the term "thread." (Response of April 14, 2005 at 7). If the meaning of the word "thread" is distorted as the examiner proposes, with no alternative word available to overcome the examiner's faulty view, there is no opportunity to claim the an invention that is fully disclosed as required by § 112 ¶¶ 1 and 2.

The examiner's "new matter" objection should be reversed.

E. The Examiner States an Incorrect Legal Test

The examiner acknowledges that the test for "new matter" permits an amendment that was "inherently contained" in the specification as filed. Action of 7/19/05 at 1, lines 14-15. Having conceded that, the examiner then states that he will only allow an amendment that is supported explicitly, by:

> "referenc[ing] those portions of the specification which supported this amended material. Therefore because applicant failed to point to the portions of the specification which support the amended material, there must be no portion of the specification which supports the material (otherwise applicant would have simply pointed to that portion) and therefore, the material is also new matter.

Action of 7/19/05 at 2, lines 1-6.

The examiner is confused. "Inherency" does not require an explicit statement that can be "pointed to;" "inherency" only comes into play when there is no explicit disclosure that could be "referenced." Because the examiner applies an incorrect legal test, the remainder of his reasoning is irrelevant at best. The examiner has made no attempt to show that the amendment is anything other than an explicit statement of material that was previously inherent.

Because the examiner's reasoning is unrelated to the correct legal test, the "objection" should be reversed.

F. The Examiner's Faulty Factual Analysis

It appears that the examiner's primary basis for his "new matter" objection is that the description added by the amendment conflicts with his extrapolated definition. The examiner's extrapolated definition is faulty, and without substantial evidence. *See American Textile Mfrs Institute Inc. v. Donovan*, 452 U.S. 490, 523 (U.S. Sup. Ct. 1981) ("substantial evidence" review requires taking into account evidence that is contrary to the agency's decision).

First, the examiner's definition of "thread" appears nowhere. His statement attributing his definition to the Microsoft dictionary, Action of 7/19/05 at page 3, lines 19-22, is simply false. The examiner relies on pure syllogistic inference, an extrapolation beyond the statements in the Microsoft dictionary. If the amendment agrees with every definition from the relevant arts that is expressly stated, including a definition cited by the examiner himself, and the only "odd man out" is the examiner's extrapolation, then it is the examiner's extrapolation that should be disregarded, not the amendment.

Second, Microsoft itself has acknowledged that the definition of "process" in the 1993 edition of the Microsoft Dictionary, on which the examiner relies, was incorrect – the definition in the current edition is significantly different.

Third, the Microsoft Dictionary was obsolete as of the filing date. Other dictionary editors and standards organizations recognized that the formal definition of "thread" and "process" had changed in the mid-1990's, and by 1996-1998 they withdrew the definitions that had been used in 1993. For example, 1996-2000 definitions of "process," from the IEEE Dictionary (Exhibit A), are as follows:

> **process (3)** An address space and one or more threads of control that execute within that address space, and their required system resources.
>
> **process (6)** An address space with one or more threads executing within that address space, and the required system resources for those threads. ... Many of the system resources defined by this part of ISO/IEC 9945 are shared among all the threads within a process. These include the process ID, the process group ID; the session membership; the real, effective and saved-set user ID; the real, effective and saved-set group ID; the supplementary group IDs; the current working directory; the root directory; the file mode creation mask; and file descriptors.
>
> **process (7)** [essentially similar to (6), in the context of the Ada programming language]

> **process (8)** An address space, a single thread of control that executes within that address space, and its required system resources. On a system that implements threads, a process is redefined to consist of an address space with one or more threads executing within that address space and their required system resources, etc. ...

The relevant definitions changed over this period. For example, the IEEE Dictionary states a definition "process (5)" from the 1993 version of ISO/IEC Std. 9945, the same year as the examiner's edition of the Microsoft Dictionary. However, definition (5) was superseded in a 1996 revision of standard ISO/IEC 9945, by definition (6). Because this application has an effective filing date of 1999, the examiner's 1993 dictionary is irrelevant.

Fourth, the portions of the Microsoft Dictionary on which the examiner relies are clearly informal. Even a cursory comparison of the Microsoft definitions cited by the examiner reveal that they are only informal definitions, barely above the level of a "general usage" dictionary – they reflect none of the characteristics that specialists actually use and rely on. The 1993 Microsoft definition of "thread" states that a thread is merely "part of" a process, without explaining any of the relevant differences and relationships. The 1993 Microsoft definition of "process" never mentions the essential attribute of a "process," its "address space." Microsoft itself discourages precise reliance on its dictionary: both the notes on the back of the current edition of the Dictionary and the Introduction state that the audience for the Microsoft Computer Dictionary is "users" and "home users," but does not mention specialists in programming language systems, operating systems, and processor architecture. The back of the Microsoft Dictionary makes clear that many definitions are over-simplified for non-specialists:

> You get simple, concise definitions for understanding even the most arcane terms..

When Microsoft writes to specialists in the art, and intends for them to rely on its descriptions, Microsoft sets out its definitions much more precisely, as set forth in Exhibit F, and quoted at page 6, above.

Just last month, in a portion of a decision joined by **eleven** of the **twelve judges of the Federal Circuit**, the Court warned against over-reliance on informal definitions, even if those informal definitions are drawn from a technical dictionary:

> For that reason, we have stated that "a general-usage dictionary cannot overcome art-specific evidence of the meaning" of a claim term. ... Even

> technical dictionaries or treatises, under certain circumstances, may suffer from some of these deficiencies. There is no guarantee that a term is used in the same way in a treatise as it would be by the patentee. ...
>
> Finally, the authors of dictionaries or treatises may simplify ideas to communicate them most effectively to the public and may thus choose a meaning that is not pertinent to the understanding of particular claim language. ... The resulting definitions therefore do not necessarily reflect the inventor's goal of distinctly setting forth his invention as a person of ordinary skill in that particular art would understand it.

Phillips v. AWH Corp., 415 F.3d 1303, 1322, 75 USPQ2d 1321, 1334 (Fed. Cir. 2005) (en banc) (underline added). Dictionaries are not infallible.

Fourth, the examiner has twice been challenged to identify any basis to believe that his extrapolation is even correct, let alone "reasonable," or "consistent with the interpretation those skilled in the art would reach." Response of 4/27/05 at 17; Response of 3/30/05 at 6-7. All he cites in support of his extrapolation is his own inference, no direct evidence. By his silence, his inability to provide any evidence in support of his own inference, his failure to rebut the uniform definitions set forth in every other source, and his failure to explain away the absurd consequences of his extrapolation (see Response of 4/27/05 at page 18 and the Response of 3/30/05 at 7), he confesses that his extrapolation is outside the scope of permitted "broadest reasonable interpretation consistent with the specification" and inconsistent with "the meaning given to the term by those of ordinary skill in the art," as required by MPEP § 2111.01.

Thus, any disagreement between the amendment and the examiner's extrapolation is no basis to conclude that the amendment is "new matter." The objection should be reversed.

II. Final Rejection is Premature

Paragraph 10 of the Office Action states that the July 2005 Action is final. This is premature.

A. Premature Final Rejection is a Petitionable Issue

The Federal Circuit[5], the Board of Appeals, and the Director have all held that petition to the Director under Rule 181 is the appropriate avenue request review of premature final

---

[5] *In re Alappat*, 33 F.3d. 1527, 1580, 31 USPQ2d 1545, 1588 (Fed. Cir. 1994) (*en banc*) (Plager, J., concurring) ("The [Director] has an obligation to ensure that all parts of the agency ... conform to official policy of the agency..."); *see also Star Fruits S.N.C. v. United States*, 393 F.3d 1277, 1284-85,

rejection, when finality violates PTO rules, and the only relief requested is reopening of prosecution.

The Board of Patent Appeals and Interferences has long held that premature closing of prosecution is never appealable. *Ex parte Fine*, 217 USPQ 76, 79 (Bd. Pat. App. 1981) (precedential) ("We are likewise not concerned with the allegedly premature nature of the final rejection... This is an administrative matter subject to petition, not a substantive matter within our jurisdiction."); *Ex parte Secor*, http://www.uspto.gov/web/offices/dcom/bpai/decisions/fd981052.pdf (BPAI 2002) (unpublished) (premature final rejection "is reviewable by petition to the Director rather than by appeal to this Board."). Petitioner has diligently sought, and has been unable to find, a single case since 1964 in which the Board reviewed an issue of premature final rejection. There are none. An agency may not require an issue to be presented to a tribunal that "lacks authority to grant the type of relief requested." *McCarthy v. Madigan*, 503 U.S. 140, 147 (1992).

Second, the Board has held that issues arising under MPEP procedures, even those relating to claims, are never appealable – the Board only reviews issues arising under the substantive portions of the Patent Act. *E.g.*, *Ex parte Haas*, 175 USPQ 217, 220 (Bd. Pat. App. 1972) ("If the examiner fails to follow the Commissioner's directions in the M.P.E.P., appellant's remedy is by way of petition to the Commissioner since this Board has no jurisdiction over the examiner's action.") (Lidoff, examiner-in-chief, concurring), *rev'd on other grounds*, 486 F.2d 1053 (CCPA 1973).

Because the issues presented in this petition, and the relief requested, are not appealable, they must be addressed when presented by Petition. 37 C.F.R. § 1.181(a)(1).

Third, the Commissioner of Patents and Trademarks (now the Director) holds that where the sole relief requested is reopening of prosecution, and the rules relied on originate with the Patent Office, instead of the statute – as in this petition – the issues are petitionable, even if the underlying issue might involve some consideration of the merits. *In re Oku*, 25 USPQ2d 1155, 1157 (Comm'r Pats and TM 1992) (emphasis supplied):

---

73 USPQ2d 1409, 1414-15 (Fed. Cir. 2005) ("petition process [is] the 'exclusive administrative check' on the discretion of examiners," to ensure that examiners act within the PTO's rules).

> The designation of a new ground of rejection, while involving a consideration of the merits, also involves the important question of whether the Board followed PTO regulations established by the [Director]....
>
> A decision to reopen prosecution ... is a question solely within the discretion of the [Director] and is in no way a review of a merits decision ...

Finally, issues are petitionable when "the rules specify that the matter is to be ... reviewed by the Director." 37 C.F.R. § 1.181(a)(2). The relevant rule is MPEP § 706.07(c), which instructs as follows:

> **706.07(c) Final Rejection, Premature**
>
> Any question as to prematureness of a final rejection ... is purely a question of practice, wholly distinct from the tenability of the rejection. It may therefore not be advanced as a ground for appeal, or made the basis of complaint before the Board of Patent Appeals and Interferences. It is reviewable by petition under 37 CFR 1.181. See MPEP § 1002.02(c).

The questions presented are within § 1.181(a)(2).

B. Examination Remains Incomplete: An Information Disclosure Statement Has Not Been Considered

The RCE filing of April 27, 2005 requested consideration of the Information Disclosure Statement submitted November 9, 2004. This IDS has not been considered. An application may not be finally rejected while examination is incomplete.

In this and several other applications, Examiner Ellis has repeatedly changed positions, applied references to different claim limitations, reinterpreted claim language, considered IDS's, etc. after "final" rejection, while closing the door against any opportunity to substantively respond to his changed positions. *See* 09/385,394, Petition of April 8, 2005. This is simply wrong, and a violation of the "fairness" that is at the heart of final rejection practice. MPEP § 706.07. Final sauce for the applicant goose is final sauce for the examiner gander. 35 U.S.C. § 3(a)(2)(A) (those who act on behalf of the Director must do so "in a fair, impartial, and equitable manner," that is, with bilateral equity, including with respect to examination under § 131). Examiner Ellis cannot have it both ways. If his work is not complete and timely, and he has to alter his position or complete his work, then prosecution may not be closed against an applicant.

C. The Application Has Not Been Examined – No Issue Has Been Developed For Appeal

In the paragraph spanning pages 2-3 of the Office Action of July 19, 2005, the examiner states that he gives no weight to the amendment because it is new matter. He then states that he examines the claims as if the amendment were already cancelled.

This is not permitted. The MPEP repeatedly instructs that the application must be examined as amended. MPEP §§ 2143.03, MPEP § 2163.06(I), 2163.07.

Even giving the examiner the greatest possible benefit of the doubt on the broadest reasonable definition of the term "thread" in absence of any definition in the specification, MPEP § 2163.07(I) and 2163.06(I) cover the facts here, and instruct that the application must now be examined under the definition stated (underline added, citations omitted):

> Mere rephrasing of a passage does not constitute new matter. Accordingly, a rewording of a passage where the same meaning remains intact is permissible. ... The mere inclusion of dictionary or art recognized definitions known at the time of filing an application would not be considered new matter. If there are multiple definitions for a term and a definition is added to the application, it must be clear from the application as filed that applicant intended a particular definition, in order to avoid an issue of new matter and/or lack of written description. ...
>
> ... The examiner should still consider the subject matter added to the claim in making rejections based on prior art since the new matter rejection may be overcome by applicant.

Here, there is no question that the description of "threads" and "processes" added to the specification reflects the "particular definition" used in the application as filed. Finality should be withdrawn, so that the application can be examined for the first time, giving full weight to the definition stated in the specification and the "broadest reasonable meaning of the words in their ordinary usage as they would be understood by one of ordinary skill in the art," as expressed in every relevant dictionary. MPEP § 2111.01.

On the record as it stands, no clear issue is developed for appeal – are the claims to be interpreted on appeal "consistently with the specification," or under the examiner's extrapolation, with the amendment ignored? Final rejection is premature. MPEP § 706.07.

When an agency employee acts short of "requirements of the applicable departmental regulations," then employee's action is "illegal and of no effect." *Vitarelli v. Seaton*, 359 U.S. 535, 545 (1959); *IMS, P.C. v Alvarez*, 129 F.3d 618, 621 (D.C. Cir. 1997) (it is a "well-settled

rule that an agency's failure to follow its own regulations is fatal to the deviant action."). The examiner's disregard of the amendment to the specification departs from agency procedures. Thus, as a matter of law, there is no rejection of this application, let alone final. Finality should be withdrawn so that the examiner will have an opportunity to examine the application.

III. Conclusion

The "new matter" objection should be reversed. "Finality" of the Office Action of July 19, 2005 should be reversed, and the examiner instructed to apply the definition set forth in the specification and the dictionaries cited above. (To avoid any doubt, this Petition does not request any adjudication on the ultimate merits of any claim, only the failure of the examiner to comply with procedural requirements for examination.)

This petition occasions no fee. Kindly charge any additional fee, or credit any surplus, to Deposit Account No. 23-2405, Order No. 114596-28-000053BS.

Respectfully submitted,

WILLKIE FARR & GALLAGHER LLP

Dated: September 15, 2005

By: _____

David E. Boundy
Registration No. 36,461

WILLKIE FARR & GALLAGHER LLP
787 Seventh Ave.
New York, New York 10019
(212) 728-8757
(212) 728-9757 Fax

Exhibits to

Petition for Review by Technology Center SPRE

A: Excerpts from The Authoritative Dictionary of IEEE Standards Terms (7th ed.)

B: American Heritage Dictionary Definition of "Thread"

C: Wikipedia Definition of "Thread"

D: Fall 2000 Lecture Slides, University of California San Diego

E: January 2000 Lecture Slides, Carnegie-Mellon University

F: Microsoft Documentation on Threads

G: Hewlett-Packard Documentation of Threads

H: Sun Microsystems Documentation for Threads

I: Digital Equipment Corp. Documentation for OpenVMS DECthreads

J: Apple Computer Documentation for Threads

K: Comparative Discussion of Several Vendors' Implementations of Threads

Exhibit A to

Petition for Review by Technology Center SPRE

Excerpts from

The Authoritative Dictionary of IEEE Standards Terms (7th ed.)

IEEE 100

THE AUTHORITATIVE DICTIONARY OF IEEE STANDARDS TERMS

SEVENTH EDITION

IEEE
Published by
Standards Information Network
IEEE Press

Trademarks and disclaimers

IEEE believes the information in this publication is accurate as of its publication date; such information is subject to change without notice. IEEE is not responsible for any inadvertent errors.

Other tradenames and trademarks in this document are those of their respective owners.

The Institute of Electrical and Electronics Engineering, Inc.
3 Park Avenue, New York, NY, 10016-5997, USA

To order IEEE Press publications, call 1-800-678-IEEE.

Print: ISBN 0-7381-2601-2 *SP1122*

See other standards and standards-related product listings at: http://standards.ieee.org/

The publisher believes that the information and guidance given in this work serve as an enhancement to users, all parties must rely upon their own skill and judgement when making use of it. The publisher does not assume any liability to anyone for any loss or damage caused by any error or omission in the work, whether such error or omission is the result of negligence or any other cause. Any and all such liability is disclaimed.

This work is published with the understanding that the IEEE is supplying information through this publication, not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought. The IEEE is not responsible for the statements and opinions advanced in this publication.

Library of Congress Cataloging-in-Publication Data

IEEE 100 : the authoritative dictionary of IEEE standards terms.—7th ed.
p. cm.
ISBN 0-7381-2601-2 (paperback : alk. paper)
1. Electric engineering—Dictionaries. 2. Electronics—Dictionaries. 3. Computer engineering—Dictionaries. 4. Electric engineering—Acronyms. 5. Electronics—Acronyms. 6. Computer engineering—Acronyms. I. Institute of Electrical and Electronics Engineers.

TK9 .I28 2000
621.3'03—dc21

00-050601

probe placement is determined by the circuit response and the circuit topology. (SCC20) 1445-1998

problem *See:* benchmark problem.

problem board In an analog computer, a removable frame of receptacles for patch cords and plugs that offers a means for interconnecting the inputs and outputs of computing elements. *See also:* patch board; patch panel. (C) 610.10-1994w, 165-1977w

problem check (analog computer) One or more tests used to assist in obtaining the correct machine solution to a problem. Static check consists of one or more tests of computing elements, their interconnections, or both, performed under static conditions. Dynamic check consists of one or more tests of computing elements, their interconnections, or both, performed under dynamic conditions. Rate test is a test that verifies that the time constants of the integrators are those selected. This term also refers to the computer-control state that implements the rate test previously described. Dynamic problem check is any dynamic check used to ascertain the correct performance of some or all of the computer components. *See also:* computer-control state. (C) 165-1977w

Problem Descriptor System (PDS/MaGen) A programming language useful in a wide variety of operations research applications, and designed to facilitate the generation of matrices and reports for mathematical programming systems. (C) 610.13-1993w

problem domain A set of similar problems that occur in an environment and lend themselves to common solutions. (C/SE) 1362-1998

problem-oriented language (1) (computers) A programming language designed for the convenient expression of a given class of problems. (MIL/C) [2], [85], [20]

(2) (software) A programming language designed for the solution of a given class of problems. Examples are list processing languages, information retrieval languages, simulation languages. (C) 610.12-1990, 610.13-1993w

problem state In the operation of a computer system, a state in which programs other than the supervisory program can execute. *Synonyms:* user state; slave state. *Contrast:* supervisor state. (C) 610.12-1990

problem variable *See:* scale factor.

procedural cohesion (software) A type of cohesion in which the tasks performed by a software module all contribute to a given program procedure, such as an iteration or decision process. *Contrast:* temporal cohesion; logical cohesion; sequential cohesion; coincidental cohesion; functional cohesion; communicational cohesion. (C) 610.12-1990

procedural interface (PI) The set of C functions used by an application and a delay and power calculation module (DPCM) to exchange information and determine the timing calculation for a design. (C/DA) 1481-1999

procedural interface function One of the C functions that comprise the DPCS procedural interface. (C/DA) 1481-1999

procedural language (1) (software) A programming language in which the user states a specific set of instructions that the computer must perform in a given sequence. All widely-used programming languages are of this type. *Synonym:* procedure-oriented language. *Contrast:* nonprocedural language. *See also:* algebraic language; list processing language; logic programming language; algorithmic language. (C) 610.12-1990

(2) A computer language in which the user states a specific set of instructions that the computer must perform in a given sequence. Examples include BASIC, COBOL, FORTRAN, and Pascal. *Synonym:* procedure-oriented language. *Contrast:* nonprocedural language. (C) 610.13-1993w

procedural programming language (software unit testing) A computer programming language used to express the sequence of operations to be performed by a computer (for example, COBOL). *See also:* nonprocedural programming language. (C/SE) 1008-1987r

procedure (1) (computers) The course of action taken for the solution of a problem. (C) [20], [85]

(2) (nuclear power quality assurance) A document that specifies or describes how an activity is to be performed. (PE/NP) [124]

(3) (A) (software) A course of action to be taken to perform a given task. (B) (software) A written description of a course of action as in definition "A;" for example, a documented test procedure. (C) (software) A portion of a computer program that is named and that performs a specific action. (C) 610.12-1990

(4) (software user documentation) Ordered series of instructions that a user follows to do one or more tasks. (C/SE) 1063-1987r

(5) (scheme programming language) A parameterized program fragment, called a subroutine or function in some programming languages. (C/MM) 1178-1990r

procedure-oriented language (1) (computers) A programming language designed for the convenient expression of procedures used in the solution of a wide class of problems. (MIL/C) [2], [20], [85]

(2) (software) *See also:* procedural language. (C) 610.12-1990

process (1) (automatic control) The collective functions performed in and by the equipment in which a variable is to be controlled. *Synonym:* controlled system. (PE/EDPG) [3]

(2) (A) (software) A sequence of steps performed for a given purpose; for example, the software development process. (B) (software) An executable unit managed by an operating system scheduler. *See also:* job; task. (C) (software) To perform operations on data. (C) 610.12-1990

(3) An address space and one or more threads of control that execute within that address space, and their required system resources. (C/PA) 14252-1996

(4) A sequence of tasks, actions, or activities, including the transition criteria for progressing from one to the next, that bring about a result. (C/SE) 1220-1994s

(5) An address space and the single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the POSIX.1 *fork*() function. The process that issues *fork*() is known as the parent process, and the new process created by the *fork*() is known as the child process. The attributes of processes required by POSIX.2 form a subset of those in POSIX.1. (C/PA) 9945-2-1993

(6) An address space with one or more threads executing within that address space, and the required system resources for those threads. A process is created by another process issuing the *fork*() function. The process that issues *fork*() is known as the parent process, and the new process created by the *fork*() is known as the child process. Many of the system resources defined by this part of ISO/IEC 9945 are shared among all of the threads within a process. These include the process ID; the parent process ID; the process group ID; the session membership; the real, effective and saved-set user ID; the real, effective and saved-set group ID; the supplementary group IDs; the current working directory; the root directory; the file mode creation mask; and file descriptors. (C/PA) 9945-1-1996

(7) An address space, and the program (including any Ada tasks contained within the program) executing within that address space, and its required system resources. A process is created by another process with procedures `POSIX_Process_Primitives.Start_Process`, `POSIX_Process_Primitives.Start_Process_Search`, or the function `POSIX_Unsafe_Process_Primitives.Fork`. The process that issues `Start_Process`, `Start_Process_Search`, or `Fork` is known as the parent process, and the newly created process is the child process. (C/PA) 1003.5-1992r

(8) An address space, a single thread of control that executes within that address space, and its required system resources. On a system that implements threads, a process is redefined to consist of an address space with one or more threads ex-

ecuting within that address space and their required system resources. *Note:* The term process is used in contrast to "system process," or the OSI usage of the term "application process."
(C/PA) 1327.2-1993w, 1224.2-1993w, 1326.2-1993w, 1328.2-1993w

(9) An organized set of activities performed for a given purpose; for example, the software development process.
(C/SE) J-STD-016-1995

(10) A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources. (C/MM) 855-1990

(11) Sequence of operations performed in and by the equipment in which a variable is to be controlled.
(SCC20) 1226-1998

(12) (A) A set of interrelated activities, which transforms inputs into outputs. *Note:* The term "activities" covers use of resources. (B) A series of actions bringing about a result.
(C/SE) 1490-1998

(13) Consists of all execution within a single distinct address space supported by the operating system of a computer.
(IM/ST) 1451.1-1999

(14) *See also:* POSIX process. (C) 1003.5-1999

Process A function that must be performed in the software life cycle. A Process is composed of Activities.
(C/SE) 1074-1995s

process architect The person or group that has primary responsibility for creating and maintaining the software life cycle process (SLCP). *See also:* software life cycle process.
(C/SE) 1074-1997

processable scored card A scored card including at least one separable part that can be processed after separation. *See also:* stub card. (C) 610.10-1994w

Process and Experiment Automation Realtime Language (PEARL) A general-purpose, high-order language designed to meet the requirements of real-time programming in process and experiment automation. (C) 610.13-1993w

Process Architect The person or group that manages the implementation of the Standard in an organization.
(C/SE) 1074.1-1995

process bound *See:* compute-bound.

process control (1) (electric pipe heating systems) The use of electric pipe heating systems to increase or maintain, or both, the temperature of fluids (or processes) in mechanical piping systems including pipes, pumps, tanks, instrumentation in nuclear power generating stations. (PE/EDPG) 622A-1984r

(2) (automatic control) Control imposed upon physical or chemical changes in a material. *See also:* feedback control system. (PE/EDPG) [3]

(3) (electric heat tracing systems) The use of electric heat tracing systems to increase or maintain, or both, the temperature of fluids (or processes) in mechanical piping systems including pipes, pumps, valves, tanks, instrumentation, etc, in power generating stations. (PE/EDPG) 622B-1988r

(4) Automatic control in which a computer is used to regulate continuous operations such as chemical processes, military operations, or manufacturing operations. *See also:* numerical control. (C) 610.2-1987

process equipment (automatic control) Apparatus with which physical or chemical changes in a material are produced. *Synonym:* plant. (PE/EDPG) [3]

process group (1) A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by a process group ID. A newly created process joins the process group of its creator. (C/PA) 9945-1-1996, 9945-2-1993

(2) A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by its process group ID. A newly created process joins the process group of its creator.
(C) 1003.5-1999

process group ID (1) The unique identifier representing a process group during its lifetime. A process group ID is a positive integer that can be contained in a *pid_t*. It shall not be reused by the system until the process group lifetime ends.
(C/PA) 9945-1-1996, 9945-2-1993

(2) A unique value identifying a process group during its lifetime. A process group ID shall not be reused by the system until the process group lifetime ends. (C) 1003.5-1999

process group leader (1) A process whose process ID is the same as its process group ID.
(C/PA) 9945-1-1996, 9945-2-1993

(2) The unique process, within a process group, that created the process group. (C) 1003.5-1999

process group lifetime (1) A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, due either to the end of the last process's process lifetime or to the last remaining process calling the *setsid*() or *setpgid*() functions.
(C/PA) 9945-1-1996

(2) A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, due either to the end of the process lifetime of the last process or to the last remaining process calling the `Set_Process_Group_ID` procedure. (C) 1003.5-1999

process ID (1) The unique identifier representing a process. A process ID is a positive integer that can be contained in a *pid_t*. A process ID shall not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID shall not be reused by the system until the process group lifetime ends. A process that is not a system process shall not have a process ID of 1.
(C/PA) 9945-1-1996, 9945-2-1993

(2) A unique value identifying a process during its lifetime. The process ID is a value of the type `Process_ID` defined in the package `POSIX_Process_Identification`. A process ID shall not be reused by the system until the process lifetime ends. In addition, if a process group exists where the process ID of the process group leader is equal to that process ID, that process ID shall not be reused by the system until the process group lifetime ends. An implementation shall reserve a value of process ID for use by system processes. A process that is not a system process shall not have this process ID.
(C) 1003.5-1999

processing *See:* multiprocessing; parallel processing; data processing; information processing.

processing cycle A single, complete execution of data processing that is periodically repeated. *Synonym:* data processing cycle. *See also:* daily cycle; monthly cycle; weekly cycle; annual cycle. (C) 610.2-1987

processing unit A functional unit that consists of one or more processors and their storage. *See also:* central processing unit.
(C) 610.10-1994w

process lifetime (1) The period of time that begins when a process is created and ends when its process ID is returned to the system. After a process is created with a *fork*() function, it is considered active. At least one thread of control and the address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a *wait*() or *waitpid*() function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends. (C/PA) 9945-1-1996

(2) A period of time that begins when a process is created and ends when its process ID is returned to the system. After a process is created, it is considered active. Its threads of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a

current density in the direction of the temperature gradient is positive. *See also:* thermoelectric device. (ED) [46]

Thomson effect The absorption or evolution of thermal energy produced by the interaction of an electric current and a temperature gradient in a homogeneous electric conductor. *Notes:* 1. An electromotive force exists between two points in a single conductor that are at different temperatures. The magnitude and direction of the electromotive force depend on the material of the conductor. A consequence of this effect is that if a current exists in a conductor between two points at different temperatures, heat will be absorbed or liberated depending on the material and on the sense of the current. 2. In a nonhomogeneous conductor, the Peltier effect and the Thomson effect cannot be separated. *See also:* thermoelectric device. (ED) [46]

Thomson heat The thermal energy absorbed or evolved as a result of the Thomson effect. *See also:* thermoelectric device. (ED) [46]

thrashing A state in which a computer sys-tem is expending most or all of its resources on overhead operations, such as swapping data between main and auxiliary storage, rather than on intended computing functions. (C) 610.12-1990

thread (1) (control) A control function that provides for maintained operation of a drive at a preset reduced speed such as for setup purposes. *See also:* electric drive. (IA/ICTL/IAC) [60]

(2) (data management) In a tree, a set of link fields, one in each node, each of which points to the successor or predecessor of that node with respect to a particular traversal order. (C) 610.5-1990w

(3) A single sequential flow of control within a process. (C/PA) 1328.2-1993w, 1326.2-1993w, 1224.2-1993w, 1327.2-1993w, 14252-1996

(4) A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, *errno* value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via *malloc*(), directly addressable storage obtained through implementation-supplied functions, and automatic variables shall be accessible to all threads in the same process. (C/PA) 9945-1-1996

threaded coupling (rigid steel conduit) An internally threaded steel cylinder for connecting two sections of rigid steel conduit. (EEC/CON) [28]

threaded tree A tree whose nodes contain link fields for one or more threads, allowing nonrecursive traversal of the tree. *See also:* left-threaded tree; triply-threaded tree; doubly-threaded tree; right-threaded tree. (C) 610.5-1990w

thread ID A unique value of type *pthread_t* that identifies each thread during its lifetime in a process. (C/PA) 9945-1-1996

threading line (conductor stringing equipment) A lightweight flexible line, normally manila or synthetic fiber rope, used to lead a conductor through the bullwheels of a tensioner or pulling line through a bull wheel puller. *Synonyms:* threading rope; bull line. (T&D/PE) 524-1992r

threading rope *See:* threading line.

thread list An ordered set of runnable threads that all have the same ordinal value for their priority. The ordering of threads on the list is determined by a scheduling policy or policies. The set of thread lists includes all runnable threads in the system. (C/PA) 9945-1-1996

thread of control A sequence of instructions executed by a conceptual sequential subprogram, independent of any programming language. More than one thread of control may execute concurrently, interleaved on a single processor, or on separate processors. The conceptual threads of control in an Ada application are Ada tasks. They may, but need not, correspond to the POSIX threads defined in POSIX.1. (C) 1003.5-1999

thread-safe A function that may be safely invoked concurrently by multiple threads. Each function defined by this standard is thread-safe unless explicitly stated otherwise. An example is any "pure" function (a function that holds a mutex locked while it is accessing static storage or objects shared among threads). (C/PA) 9945-1-1996

thread-specific data key A process global handle of type *pthread_key_t* that is used for naming thread-specific data. Although the same key value may be used by different threads, the values bound to the key by *pthread_setspecific*() and accessed by *pthread_getspecific*() are maintained on a per-thread basis and persist for the life of the calling thread. (C/PA) 9945-1-1996

threat (1) A potential violation of security. (LM/C) 802.10g-1995, 802.10-1992

(2) Means by which a system may be adversely affected. Threats include both inadvertent and malicious actions. (C/BA) 896.3-1993w

three-address Pertaining to an instruction code in which each instruction has three address parts. Also called triple-address. In a typical three-address instruction the addresses specify the location of two operands and the destination of the result, and the instructions are taken from storage in a preassigned order. *See also:* two-plus-one address. (C) 162-1963w

three-address instruction (1) A computer instruction that contains three address fields. For example, an instruction to add the contents of locations A and B, and place the results in location C. *Contrast:* four-address instruction; one-address instruction; zero-address instruction; two-address instruction. (C) 610.12-1990

(2) An instruction containing three addresses. *Synonym:* triple-address instruction. *See also:* address format. (C) 610.10-1994w

three-bit byte *See:* triplet.

three-conductor bundle *See:* bundle.

three-dimensional graphics The presentation of data on a two-dimensional display surface so that it appears to represent a three-dimensional model, and can be viewed from any position. *Note:* Each coordinate of the model contains a triplet of information; for example, x, y, and z in the Cartesian coordinate system. (C) 610.6-1991w

three-dimensional hardware A graphical display processor that accepts three-dimensional information as input and generates an image directly rather than using a projection transformation. (C) 610.6-1991w

three-dimensional priority The property possessed by a line or surface that is in front of another line or surface from the viewer's perspective. (C) 610.6-1991w

three-dimensional radar (navigation aid terms) A radar capable of producing three-dimensional position data on a multiplicity of targets. (AES/GCS) 686-1997, 172-1983w

3GL *See:* high-order language.

three-input adder *See:* full adder.

three-level address *See:* n-level address.

3-of-9 bar code A variable length, bidirectional, discrete, self-checking, alpha-numeric bar code. Its basic data character set contains 43 characters: 0 to 9, A to Z, −, ., /, +, $, %, and space. Each character is composed of 9 elements: 5 bars and 4 spaces. Three of the nine elements are wide (binary value 1) and six are narrow (binary value 0). A common character (*) is used exclusively for both a start and stop character. (PE/TR) C57.12.35-1996

three-phase ac fields (electric and magnetic fields from ac power lines) Three-phase transmission lines generate a three-phase field whose space components are not in phase. The field at any point can be described by the field ellipse, that is, by the magnitude and direction of the semi-major axis and the magnitude and direction of its semi-major axis. In a three-phase field, the electric field at large distances ≥ 15 m away from the outer phases (conductors) can frequently be considered a single-phase field because the minor axis of the electric

Exhibit B to

Petition for Review by Technology Center SPRE

American Heritage Dictionary Definition of "Thread"

THE AMERICAN HERITAGE® DICTIONARY OF THE ENGLISH LANGUAGE

THIRD EDITION

HOUGHTON MIFFLIN COMPANY
Boston · New York

Houghton Mifflin Company gratefully acknowledges Mead Data Central, Inc., providers of the LEXIS®/NEXIS® services, for its assistance in the preparation of this edition of *The American Heritage® Dictionary*.

Library of Congress Cataloging-in-Publication Data

The American heritage dictionary of the English language.
—3rd ed.
p. cm.
ISBN 0-395-44895-6
1. English language—Dictionaries.
PE1628.A623 1992 92-851
423—dc20 CIP

Manufactured in the United States of America

For information about this and other Houghton Mifflin trade and reference books and multimedia products, visit The Bookstore at Houghton Mifflin on the World Wide Web at http://www.hmco.com/trade/.

thread (thrĕd) *n.* **1.a.** Fine cord of a fibrous material, such as cotton or flax, made of two or more filaments twisted together and used in needlework and the weaving of cloth. **b.** A piece of such cord. **2.a.** A thin strand, cord, or filament of natural or manufactured material. **b.** Something that suggests the fineness or thinness of such a strand, cord, or filament: *a thread of smoke.* **c.** Something that suggests the continuousness of such a strand, cord, or filament: *lost the thread of his argument.* **3.** A helical or spiral ridge on a screw, nut, or bolt. **4. threads.** *Slang.* Clothes. —**thread** *v.* **thread·ed, thread·ing, threads.** —*tr.* **1.a.** To pass one end of a thread through the eye of (a needle, for example). **b.** To pass (something) through in the manner of a thread: *thread the wire through the opening.* **c.** To pass a tape or film into or through (a device): *thread a film projector.* **d.** To pass (a tape or film) into or through a device. **2.** To connect by running a thread through; string: *thread beads.* **3.a.** To make one's way cautiously through: *threading dark alleys.* **b.** To make (one's way) cautiously through something. **4.** To occur here and there throughout; pervade: *"More than 90 geologic faults thread the Los Angeles area"* (Science News). **5.** To machine a thread on (a screw, nut, or bolt). —*intr.* **1.** To make one's way cautiously: *threaded through the shoals and sandbars.* **2.** To proceed by a winding course. **3.** To form a thread when dropped from a spoon, as boiling sugar syrup. [Middle English, from Old English *thrǣd.* See **tere-**[1] in Appendix.] —**thread′er** *n.*

thread·bare (thrĕd′bâr′) *adj.* **1.** Having the nap worn down so that the filling or warp threads show through; frayed or shabby: *threadbare rugs.* **2.** Wearing old, shabby clothing. **3.** Overused to the point of being worn out; hackneyed: *threadbare excuses.* See Synonyms at **trite.**

thread·fin (thrĕd′fĭn′) *n., pl.* **threadfin** or **-fins.** Any of various chiefly tropical marine fishes of the family Polynemidae, having threadlike rays extending from the lower part of the pectoral fin.

thread·worm (thrĕd′wûrm′) *n.* See **pinworm.**

thread·y (thrĕd′ē) *adj.* **-i·er, -i·est. 1.** Consisting of or resembling thread; filamentous. **2.** Capable of forming or tending to form threads; viscid. **3.** *Medicine.* Weak and shallow. Used of a pulse. **4.** Lacking fullness of tone; thin: *a thready voice.* —**thread′i·ness** *n.*

Exhibit C to

Petition for Review by Technology Center SPRE

Wikipedia Definition of "Thread"

Thread (computer science)

From Wikipedia, the free encyclopedia.

A **thread** in computer science is short for a *thread of execution* or a sequence of instructions. Multiple threads can be executed in parallel on many computer systems. This *multithreading* generally occurs by time slicing (where a single processor switches between different threads) or by multiprocessing (where threads are executed on separate processors). Threads are similar to processes, but differ in the way that they share resources.

Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The operating system kernel allows programmers to manipulate threads via the system call interface. Some implementations are called *kernel threads* or *lightweight processes*.

Absent that, programs can still implement threading by using timers, signals, or other methods to interrupt their own execution and hence perform a sort of ad-hoc time-slicing. These are sometimes called *user-space threads*.

Threads are a way for a program to split itself into two or more simultaneously running tasks. (The name "thread" is by analogy with the way that a number of threads are interwoven to make a piece of fabric).

A common use of threads is having one thread paying attention to the graphical user interface, while others do a long calculation in the background. As a result, the application more readily responds to user's interaction.

An unrelated use of the term **thread** is for threaded code, which is a form of code consisting entirely of subroutine calls, written without the subroutine call instruction, and processed by an interpreter or the CPU. Two threaded code languages are Forth and early B programming languages.

Contents

Threads compared with processes

Threads are distinguished from traditional multi-tasking operating system processes in that processes are typically independent, carry considerable state information, have separate address spaces, and interact only through system-provided inter-process communication mechanisms. Multiple threads, on the other hand, typically share the state information of a single process, and share memory and other resources directly. Context switching between threads in the same process is typically faster than context switching between processes. Systems like Windows NT and OS/2 are said to have "cheap" threads and "expensive" processes, while in other operating systems there is not so big a difference.

An advantage of a multi-threaded program is that it can operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines. This is because the threads of the program naturally lend themselves for truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require atomic operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such

primitives can lead to deadlocks.

Operating systems generally implement threads in one of two ways: preemptive multithreading, or cooperative multithreading. Preemptive multithreading is generally considered the superior implementation, as it allows the operating system to determine when a context switch should occur. Cooperative multithreading, on the other hand, relies on the threads themselves to relinquish control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available. The disadvantage to preemptive multithreading is that the system may make a context switch at an inappropriate time, causing priority inversion or other bad effects which may be avoided by cooperative multithreading.

Traditional mainstream computing hardware did not have much support for multithreading as switching between threads was generally already quicker than full process context switches. Processors in embedded systems, which have higher requirements for real-time behaviors, might support multithreading by decreasing the thread switch time, perhaps by allocating a dedicated register file for each thread instead of saving/restoring a common register file. In the late 1990s, the idea of executing instructions from multiple threads simultaneously has become known as simultaneous multithreading. This feature was introduced in Intel's Pentium 4 processor, with the name *Hyper-threading*.

Processes, threads, and fibers

The concept of a *process*, *thread*, and *fiber* are interrelated by a sense of "ownership" and of containment.

A *process* is the "heaviest" unit of kernel scheduling. Processes own resources allocated by the operating system. Resources include memory, file handles, sockets, device handles, and windows. Processes do not share address spaces or file resources except through explicit methods such as inheriting file handles or shared memory segments, or mapping the same file in a shared way. Processes are typically pre-emptively multitasked. However, Windows 3.1 and older versions of Mac OS used co-operative or non-preemptive multitasking.

A *thread* is the "lightest" unit of kernel scheduling. At least one thread exists within each process. If multiple threads can exist within a process, then they share the same memory and file resources. Threads are pre-emptively multitasked if the operating system's process scheduler is pre-emptive. Threads do not own resources except for a stack and a copy of the registers including the program counter.

In some situations, there is a distinction between "kernel threads" and "user threads" -- the former are managed and scheduled by the kernel, whereas the latter are managed and scheduled in userspace. In this article, the term "thread" is used to refer to kernel threads, whereas "fiber" is used to refer to user threads. Fibers are co-operatively scheduled: a running fiber must explicitly "yield" to allow another fiber to run. A fiber can be scheduled to run in any thread in the same process.

Thread and fiber issues

Typically fibers are implemented entirely in userspace. As a result, context switching between fibers in a process is extremely efficient: because the kernel is oblivious to the existence of fibers, a context switch does not require any interaction with the kernel at all. Instead, a context switch can be performed by locally saving the CPU registers used by the currently executing fiber and loading the registers required by the fiber to be executed. Since scheduling occurs in userspace, the scheduling policy can be more easily tailored to the requirements of the program's workload.

However, the use of blocking system calls in fibers can be problematic. If a fiber performs a system call that blocks , the other fibers in the process are unable to run until the system call returns. A typical example of this problem is when performing I/O: most programs are written to perform I/O synchronously. When an I/O operation is initiated, a system call is made, and does not return until the I/O operation has been completed. In the intervening period, the entire process is "blocked" by the kernel, and cannot run -- which starves other fibers in the same process from executing.

A common solution to this problem is providing an I/O API that implements a synchronous interface by using non-blocking I/O internally, and scheduling another fiber while the I/O operation is in progress. Similar solutions can be provided for other blocking system calls. Alternatively, the program can be written to avoid the use of synchronous I/O or other blocking system calls.

Win32 supplies a fiber API. SunOS 4.x implemented "light-weight processes" or LWPs as fibers known as "green threads". SunOS 5.x and later, NetBSD 2.x, and DragonFly BSD implement LWPs as threads as well.

The use of kernel threads brings simplicity; the program doesn't need to know how to manage threads, as the kernel handles all aspects of thread management. There are no blocking issues since if a thread blocks, the kernel can reschedule another thread from within the process or from another, nor are extra system calls needed.

However, there are obvious issues with managing threads through the kernel, since on creation and removal, a context switch between kernel and usermode needs to occur, so programs that rely on using a lot of threads for short periods may suffer performance hits.

Hybrid schemes are available which provide a tradeoff between the two.

Relationships between processes, threads, and fibers

The operating system creates a process for the purpose of running a program. Every process has at least one thread. On some operating systems, processes can have more than one thread. A thread can use fibers to implement cooperative multitasking to divide the thread's CPU time for multiple tasks. Generally, this is not done because threads are cheap, easy, and well implemented in modern operating systems.

Processes are used to run an instance of a program. Some programs like word processors are designed to have only one instance of themselves running at the same time. Sometimes, such programs just open up more windows to accommodate multiple simultaneous use. After all, you can go back and forth between five documents, but you can edit one of them at a given instance.

Other programs like command shells maintain a state that you want to keep separate. Each time you open a command shell in Windows, the operating system creates a process for that shell window. The shell windows do not affect each other. Some operating systems support multiple users being logged in simultaneously. It is typical for dozens or even hundreds of people to be logged into some Unix systems. Other than the sluggishness of the computer, the individual users are (usually) blissfully unaware of each other. If Bob runs a program, the operating system creates a process for it. If Alice then runs the same program, the operating system creates another process to run Alice's instance of that program. So if Bob's instance of the program crashes, Alice's instance does not. In this way, processes protect users from failures being experienced by other users.

However, there are times when a single process needs to do multiple things concurrently. The quintessential example is a program with a graphical user interface (GUI). The program must repaint its GUI and respond to user interaction even if it is currently spell-checking a document or playing a song. For situations like these, threads are used.

Threads allow a program to do multiple things concurrently. Since the threads, spawned by a program, share the same address space, one thread can modify data that is used by another thread. This is both a good and a bad thing. It is good because it facilitates easy communication between threads. It can be bad because a poorly written program may cause one thread to inadvertently overwrite data being used by another thread. The sharing of a single address space between multiple threads is one of the reasons that multithreaded programming is usually considered to be more difficult and error-prone than programming a single-threaded application.

There are other potential problems as well such as deadlocks, livelocks, and race conditions. However, all of these problems are concurrency issues and as such affect multi-process and multi-fiber models as well.

Threads are also used by web servers. When a user visits a web site, a web server will use a thread to serve the page to that user. If another user visits the site while the previous user is still being served, the web server can serve the second visitor by using a different thread. Thus, the second user does not have to wait for the first visitor to be served. This is very important because not all users have the same speed Internet connection. A slow user should not delay all other visitors from downloading a web page. For better performance, threads used by web servers and other Internet services are typically pooled and reused to eliminate even the small overhead associated with creating a thread.

Fibers were popular before threads were implemented by the kernels of operating systems. Historically, fibers can be thought of as a trial run at implementing the functionality of threads. There is little point in using fibers today because threads can do everything that fibers can do and threads are implemented well in modern operating systems.

Many software developers believe that in most cases attempts to use fibers actually decrease the performance of an application. There are several reasons for this. First, the use of fibers does not increase the percentage of CPU time that the operating sytem gives to a process. Second, in typical multithreaded or multifibered code there is contention for shared resources like variables.

Programming constructs like semaphores and critical sections are used to solve problems that multithreading and multifibering create. Often these tools are implemented more efficiently in the operating system than in user libraries, particularly if those libraries are not written in assembly language. Third, as users install newer versions of an operating system, improvements may be made to the kernel scheduler, operating system provided semaphores, etc. User libraries do not see these benefits unless they call libraries provided by the operating system.

A case where fibers still can be helpful is when the limits of the OS in terms of number of threads per process are reached (for example 2000-3000 threads). In this case context-switching, which includes a system call, is too expensive and fibers can help. This situation may happen, for example, in a server that spawns a new thread for each incoming request.

Implementations

There are many different and incompatible implementations of threading. These can either be kernel-level or user-level implementations.

Note that fibers can be implemented without operating system support, although some operating systems or libraries provide explicit support for them. For example, recent versions of Microsoft Windows (Windows NT 3.51 SP3 and later) support a fiber API for applications that want to gain performance improvements by managing scheduling themselves, instead of relying on the kernel scheduler (which may not be tuned for the application). Microsoft SQL Server 2000's user mode scheduler, running in fiber mode, is an example of doing this.

Kernel-level implementation examples

- LWKT in various BSDs
- M:N threading (in BSDs)
- NPTL Native Posix Threading Library for Linux from Red Hat.

User-level implementation examples

- FSU Pthreads
- LWP

Comparison between models

An operating system can provide support for processes, threads, and fibres in any combination; which of these it provides will have a significant impact on its overall behavior and characteristics.

In the table below, P stands for processes, T stands for threads, and F stands for fibers.

P	T	F	Example
✗ No	✗ No	✗ No	A program running on DOS. The program can only do one thing at a time.
✗ No	✗ No	✓ Yes	Windows 3.1 running on top of DOS. All Windows programs are run within a single process, so the programs can corrupt each other's memory space. A poorly written program could corrupt data it didn't own, causing the infamous General Protection Fault.
✗ No	✓ Yes	✗ No	Amiga OS's original implementation. The operating system had full thread support, allowing multiple applications to run independently of each other which are scheduled by the kernel. The lack of process support resulted in a more efficient system (by avoiding the overhead of memory protection), with the price that application bugs could easily crash the entire computer.
✗ No	✓ Yes	✓ Yes	This case is used only in embedded systems and small real-time operating systems. Theoretically possible in a general purpose operating system, but no known examples.
✓ Yes	✗ No	✗ No	Most early implementations of Unix. The operating system could run more than one program at a time, and programs were protected from each other. If a program behaved badly, it could crash its process ending that instance of the program without disrupting the operating system or other programs. However, due to this protection, sharing information between programs was error-prone (using techniques like shared memory) and expensive (using techniques like message passing). Performing any

			tasks asynchronously required an expensive fork() system call.
✓ Yes	✗ No	✓ Yes	Sun OS before Solaris. Sun OS is Sun Microsystem's version of Unix. Sun OS implemented "green threads" in order to allow a single process to asynchronously perform multiple tasks such as playing a sound, repainting a window, and responding to user events such as clicking the stop button. Although processes were pre-emptively scheduled, the "green threads" or fibers were co-operatively multitasked. Often this model was used before real threads were implemented. This model is still used in microcontrollers and embedded devices.
✓ Yes	✓ Yes	✗ No	This is the most common case for applications running on Windows NT, Windows 2000, Windows XP, Mac OS X, Linux, and other modern operating systems. Although each of these operating systems allows the programmer to implement fibers or use a fiber library, most programmers do not use fibers in their applications. The programs are multithreaded and run inside a multitasking operating system, but perform no user-level context switching. On the typical home computer, most running processes have two or more threads. A few processes will have a single thread. Usually these processes are services running without user interaction. Typically there are no processes using fibers.
✓ Yes	✓ Yes	✓ Yes	Almost all operating systems after 1995 fall into this category. The use of threads to perform concurrent operations is the most common choice, although there are also multi-process and multi-fiber applications. Threads are used, for example, to enable a program to render its graphical user interface while waiting for input from the user or performing a task like spell checking.

Example of multithreaded code

This is an example of a simple multi-threaded program written in C#. The program illustrates the use of threads and a possible outcome of executing it on the Linux platform. Note, the outcome is not fixed and these threads could potentially lead to a "race situation."

```
using System;
using System.Threading;
public class ThreadWork {
  public static void DoWork() {
    for (int i = 0; i<3;i++) {
         Console.WriteLine("Worker thread ...");
         Thread.Sleep(100);
    }
  }
}
class ThreadTest{
  public static void Main() {
    ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
    Thread myThread = new Thread(myThreadDelegate);
    myThread.Start();
    for (int i = 0; i<3; i++) {
        Console.WriteLine("In main.");
        Thread.Sleep(100);
    }
  }
}
```

One possible set of output is:

```
In main.
Worker thread ...
In main.
Worker thread ...
In main.
Worker thread ...
```

See also

- List of multi-threading libraries
- clone()

- Communicating sequential processes
- completion port
- computer multitasking
- fork()
- Lock-free and wait-free algorithms
- message passing
- priority inversion
- synchronization
- thread safety

References

- David R. Butenhof: *Programming with POSIX Threads*, Addison-Wesley, ISBN 0-201-63392-2
- Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farell: *Pthreads Programming*, O'Reilly & Associates, ISBN 1-56592-115-1
- Charles J. Northrup: *Programming with UNIX Threads*, John Wiley & Sons, ISBN 0-471-13751-0
- Mark Walmsley: *Multi-Threaded Programming in C++*, Springer, ISBN 1-85233-146-1
- Paul Hyde: *Java Thread Programming*, Sams, ISBN 0-672-31585-8
- Bill Lewis: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, ISBN 0-1-3443698-9
- Steve Kleiman, Devang Shah, Bart Smaalders: *Programming With Threads*, SunSoft Press, ISBN 0-13-172389-8
- Pat Villani: *Advanced WIN32 Programming: Files, Threads, and Process Synchronization*, Harpercollins Publishers, ISBN 0-87930-563-0
- Jim Beveridge, Robert Wiener: *Multithreading Applications in Win32*, Addison-Wesley, ISBN 0-201-44234-5
- Thuan Q. Pham, Pankaj K. Garg: *Multithreaded Programming with Windows NT*, Prentice Hall, ISBN 0-131-20643-5
- Len Dorfman, Marc J. Neuberger: *Effective Multithreading in OS/2*, McGraw-Hill Osborne Media, ISBN 0070178410
- Alan Burns, Andy Wellings: *Concurrency in ADA*, Cambridge University Press, ISBN 0-521-62911-X
- Uresh Vahalia: *Unix Internals: the New Frontiers*, Prentice Hall, ISBN 0-13-101908-2
- Alan L. Dennis: *.Net Multithreading* , Manning Publications Company, ISBN 1-930-11054-5
- Tobin Titus, Fabio Claudio Ferracchiati, Srinivasa Sivakumar, Tejaswi Redkar, Sandra Gopikrishna: *C# Threading Handbook*, Peer Information Inc, ISBN 1-861-00829-5
- Tobin Titus, Fabio Claudio Ferracchiati, Srinivasa Sivakumar, Tejaswi Redkar, Sandra Gopikrishna: *Visual Basic .Net Threading Handbook*, Wrox Press Inc, ISBN 1-861-00713-2

External links

- Real World Tech article by Paul DeMone (http://www.realworldtech.com/page.cfm?ArticleID=RWT122600000000) - Explaining different types of multithreading, hardware implementation requirements and the impact on software.
- Ars Technica article about multithreading, etc (http://arstechnica.com/paedia/h/hyperthreading/hyperthreading-1.html)
- Page 1 (http://www.ece.utexas.edu/~valvano/EE345M/view04.pdf) & Page 2 (http://www.ece.utexas.edu/~valvano/EE345M/view05.pdf), a preemptive multithreaded implementation described
- Forum (news:comp.programming.threads)
- Answers to frequently asked questions for comp.programming.threads (http://www.serpentine.com/~bos/threads-faq/)
- Frequently Asked Questions (http://lambdacs.com/cpt/FAQ.html), Most FAQ (http://lambdacs.com/cpt/MFAQ.html)
- Discussion "Writing a scalable server" (http://groups.google.com/groups?group=comp.programming.threads&threadm=580fae16.0312210310.1410bf2b%40posting.google.com)
- The C10K problem (http://www.kegel.com/c10k.html)
- Business logic processing in a socket server - thread pools (http://www.jetbyte.com/portfolio-showarticle.asp?articleId=38&catId=1&subcatId=2)
- System support for scalable network servers (http://www.cs.rice.edu/CS/Systems/ScalaServer/)
- cohort scheduling (http://citeseer.nj.nec.com/larus02using.html)
- Article "Multi-threading basics (http://www.niccolai.ws/works/articoli/art-multithreading-en-1a.html)" by Giancarlo Niccolai
- Article "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software (http://gotw.ca/publications/concurrency-ddj.htm)" by Herb Sutter
- An Implementation of Scheduler Activations on the NetBSD Operating System (http://web.mit.edu/nathanw/www/usenix/)
- DragonFly - Light Weight Kernel Threading Model (http://www.dragonflybsd.org/goals/threads.cgi)
- Java(TM) and Solaris(TM) Threading (http://java.sun.com/docs/hotspot/threads/threads.html)
- Article "The Object-Oriented Amiga Exec (http://cunningham-lee.com/misc/amiga_exec.html)" by Tim Holloway (BYTE Magazine, January 1991)

- AROS Programmers Reference Book (http://www.falvotech.com/book.html)

Retrieved from "http://en.wikipedia.org/wiki/Thread_%28computer_science%29"

Categories: Operating system technology | Computer architecture

Exhibit D to

Petition for Review by Technology Center SPRE

Fall 2000 Lecture Slides, University of California San Diego

CSE 120
Principles of Operating Systems

Fall 2000

Lecture 5: Threads

Geoffrey M. Voelker

Processes

- Recall that a process includes many things
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)
- Creating a new process is costly because of all of the data structures that must be allocated and initialized
  - FreeBSD: 81 fields, 408 bytes
  - ...which does not even include page tables, etc.
- Communicating between processes is costly because most communication goes through the OS
  - Overhead of system calls and copying data

Parallel Programs

- Also recall our Web server example that forks off copies of itself to handle multiple simultaneous requests
  - Or any parallel program that executes on a multiprocessor
- To execute these programs we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
    - » They are all part of the same computation
  - Have the OS schedule these processes in parallel (logically or physically)
- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.

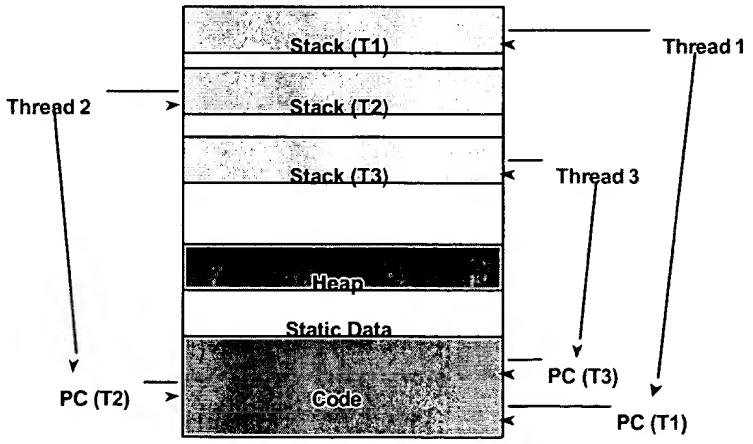
Rethinking Processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don't they share?
  - Each has its own execution state: PC, SP, and registers
- Key idea: Why don't we separate the concept of a process from its execution state?
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers
- Exec state also called thread of control, or thread

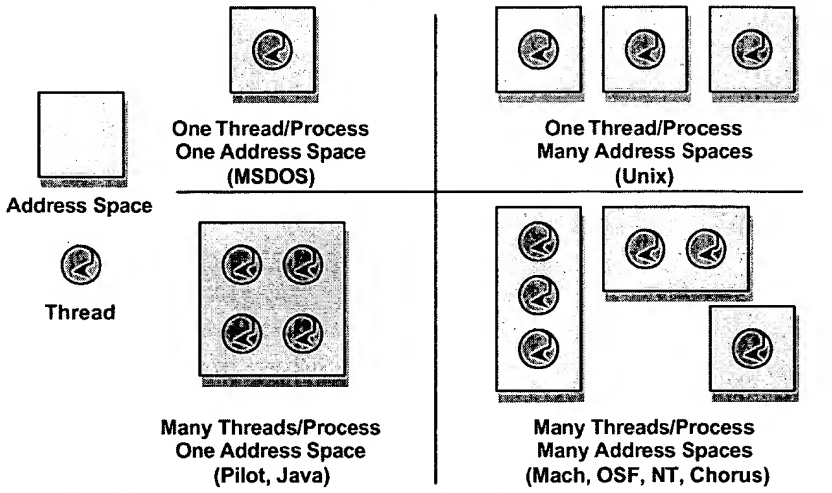
Threads

- Modern OSes (Mach, Chorus, NT, modern Unix) separate the concepts of processes and threads
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities

Threads in a Process



Thread Design Space



Process/Thread Separation

- Separating threads and processes makes it easier to support multithreaded applications
  - Creating concurrency does not require creating new processes
- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs
- So multithreading is even useful on a uniprocessor

Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task
- Recall our forking Web server:

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    Handle client request
  } else {
    Close socket
  }
}
```


Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```
web_server() {
   while (1) {
     int sock = accept();
     thread_fork(handle_request, sock);
   }
}

handle_request(int sock) {
     Process request
     close(sock);
}
```

Kernel-Level Threads

- We have taken the execution aspect of a process and separated it out into threads
  - To make concurrency cheaper
- As such, the OS now manages threads *and* processes
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
- OS-managed threads are called kernel-level threads or lightweight processes
  - NT: threads
  - Solaris: lightweight processes (LWP)

Kernel Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
  - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
  - Thread operations still require system calls
    - » Want thread operations to be as fast as a procedure call
  - Kernel-level threads have to be general to support the needs of all programmers, languages, runtimes, etc.
- For such fine-grained concurrency, need even "cheaper" threads

User-Level Threads

- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call (no kernel involvement)
  - User-level thread operations 100x faster than kernel threads

U/L Thread Limitations

- But, user-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - Unscheduling a process with a thread holding a lock
- Solving this requires communication between the kernel and the user-level thread manager

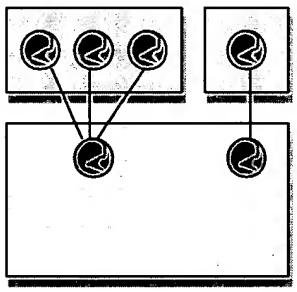
Kernel vs. User Threads

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize
- User-level threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel and user-level threads is important
  - For programming (correctness, performance)
  - For test-taking

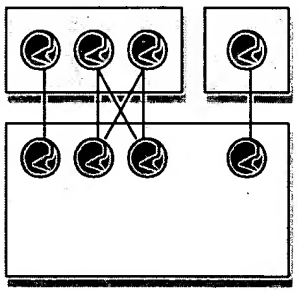
Kernel and User Threads

- Another possibility is to use both kernel and user-level threads
  - Can associate a user-level thread with a kernel-level thread
  - Or, multiplex user-level threads on top of kernel-level threads
- Java Virtual Machine (JVM)
  - Java threads are user-level threads
  - On older Unix, only one "kernel thread" per process
    - » Multiplex all Java threads on this one kernel thread
  - On NT, Solaris, OSF
    - » Can multiplex Java threads on multiple kernel threads
    - » Can have more Java threads than kernel threads
    - » Why?

User and Kernel Threads



Multiplexing user-level threads on a single kernel thread for each process



Multiplexing user-level threads on multiple kernel threads for each process

Implementing Threads

- Implementing threads has a number of issues
  - Interface
  - Context switch
  - Preemptive vs. non-preemptive
  - Scheduling
  - Synchronization (next lecture)
- Focus on user-level threads
  - Kernel-level threads are similar to original process management and implementation in the OS
  - What you will be dealing with in Nachos
  - Not only will you be *using* threads in Nachos, you will be *implementing* more thread functionality

Thread Interface

- thread_fork(procedure_t)
  - Create a new thread of control
  - Also thread_create(), thread_setstate()
- thread_stop()
  - Stop the calling thread; also thread_block
- thread_start(thread_t)
  - Start the given thread
- thread_yield()
  - Voluntarily give up the processor
- thread_exit()
  - Terminate the calling thread; also thread_destroy

Thread Scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - But it is implemented at user-level in a library
- Run queue: Threads currently running (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?
  - How would you implement thread_sleep(time)?

Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with thread_yield

Ping Thread

```
while (1) {
    printf("ping\n");
    thread_yield();
}
```

Pong Thread

```
while (1) {
    printf("pong\n");
    thread_yield();
}
```

- What is the output of running these two threads?

thread_yield()

- Wait a second. How does thread_yield() work?
- The semantics of thread_yield are that it gives up the CPU to another thread
  - In other words, it context switches to another thread
- So what does it mean for thread_yield to return?
  - It means that *another thread* called thread_yield!
- Execution trace of ping/pong
  - printf("ping\n");
  - thread_yield();
  - printf("pong\n");
  - thread_yield();
  - ...

Implementing thread_yield()

```
thread_yield() {
   thread_t old_thread = current_thread;
   current_thread = get_next_thread();
   append_to_queue(ready_queue, old_thread);
   context_switch(old_thread, current_thread);
   return;
}
```

As old thread

As new thread

- The magic step is invoking context_switch()
- Why do we need to call append_to_queue()?

Thread Context Switch

- The context switch routine does all of the magic
  - Saves context of the currently running thread (old_thread)
    - » Push all machine state onto its stack (*not* its TCB)
  - Restores context of the next thread
    - » Pop all machine state from the next thread's stack
  - The next thread becomes the current thread
  - Return to caller as new thread
- This is all done in assembly language
  - It works **at** the level of the procedure calling convention, so it cannot be implemented using procedure calls
  - See code/threads/switch.s in Nachos

Preemptive Scheduling

- Non-preemptive threads have to voluntarily give up CPU
  - A long-running thread will take over the machine
  - Only voluntary calls to thread_yield(), thread_stop(), or thread_exit() causes a context switch
- Preemptive scheduling causes an involuntary context switch
  - Need to regain control of processor asynchronously
  - Use timer interrupt
  - Timer interrupt handler forces current thread to "call" thread_yield
    - » How do you do this?
  - Nachos is preemptive
    - » See use of thread->yieldOnReturn in code/machine/interrupt.cc

Threads Summary

- The operating system is a large multithreaded program
  - Each process executes as a thread within the OS
- Multithreading is also very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization...

Next time...

- Read Chapter 8: 8.6-8.23
- Project #1 out

Exhibit E to

Petition for Review by Technology Center SPRE

January 2000 Lecture Slides, Carnegie-Mellon University

Return to the Lecture Notes Index

Lecture 4 (January 24, 2000)

Reading

Chapters 5 & 6

The Process Control Block (PCB)

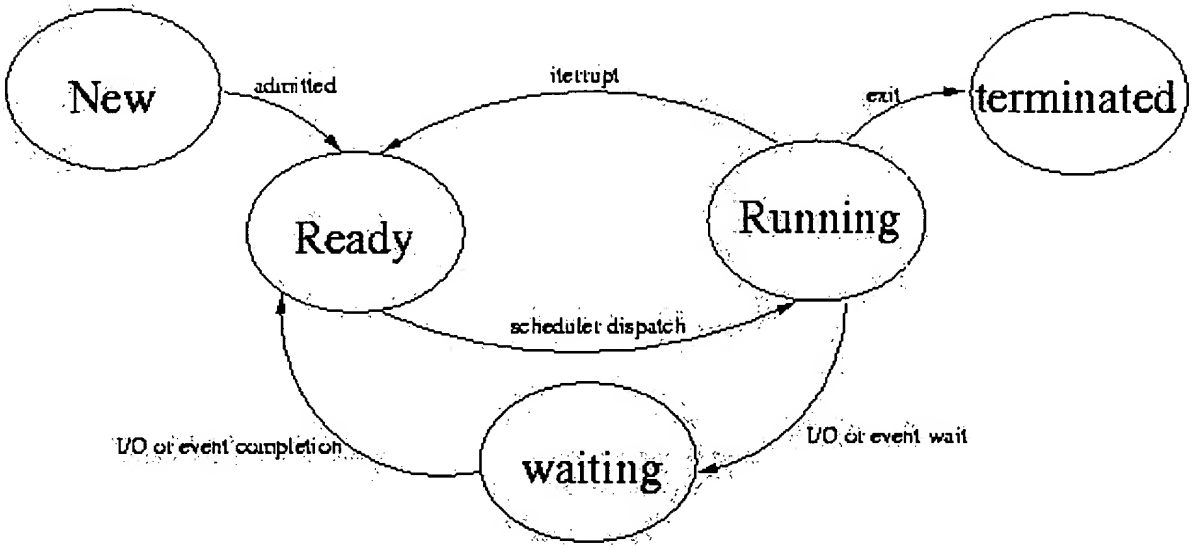
We've already discussed several different types of hardware state that are associated with a process. In addition to the hardware-context, there is also the software-context of the process. This includes the state of the programs memory as well as the information that the operating systems maintains about each process. This information is stored in an operating system structure called the *process control block* (pcb). Among other things, the PCB contains the following:

- The *process ID* of the process - a unique number that identifies or names the process within the operating system
- The group ID - a number that identifies the group or classification of users to which the process belongs.
- Information about open files
- Accounting information (CPU time used, bytes read/written, &c)
- Current state (BLOCKED, READY, RUNNING) (more later)
- Linked lists and queue pointers (more later)
- Exit status that is maintained for wait (more later)

Student Question: Does the PCB maintain scheduling information?
Answer: Yes. It contains the process state, and perhaps accounting information that will affect its scheduling priority. We'll talk more about scheduling soon.

Process State Diagram

You saw this diagram last class:



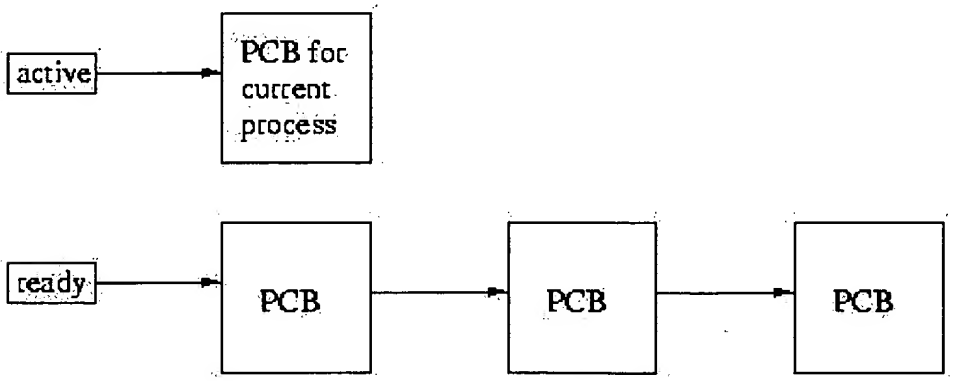
When a process exits, most of the process's resources are deleted. But the exit status is maintained in the PCB until the parent picks up the status and deletes the PCB. A process is said to be a *zombie*, if the parent dies before it does. In this case, the init process will wait for the child and delete the PCB.

There is a global variable, *active*, that keeps track of which process is currently running.

The Ready Queue

In most systems there is only one process, in others there are several procressors. In general, we assume that there can be many more processes that are ready to run than can actually run at any point in time.

For this reason, we maintain a queue of ready processes. Our favorite way of implementing a queue is with a doubly linked list. But there many different ways. In class, for simplicity, We'll just draw queues as singly linked lists.



One simple *scheduling disciple* involves selecting the active process by moving back and forth, round robin, through the ready list. Each process is selected, in turn, to become the active process.

More about scheduling soon.

The Blocked Queue

Generically, we can view the blocked state as consisting of a single *blocked queue,* containing all processes that are blocked -- regardless of the reason. But actual systems implement a separate queue for each reason.

For example, there is a separate queue for each terminal device, a separate queue for each process's inbound IPC, and a separate queue for each process's inbound IPC communications.

Lots of Pointers

More sophisticated systems might have multiple ready queues in addition ot the multiple wait queues. Different ready queues could be used to contain different types of ready processes.

It is also difficult to find the PCB from a PID, since there are many queues to search.

The PCB for a process can actually be in multiple queues at the same time.

The consequence is that real operating systems have lots of ugly pointers to implement these interconnected queues and other data structures. We'll draw them separately and neatly, but that is only so that the examples are clear.

Switching Contexts

Schedulers often change the running process. This is process requires a *context switch.* As you might conclude by considering the amount of process-state that must be saved and restored, this is a very expensive process.

So why would an operating system want to context switch?

- Allow one proces to run while another is waiting for I/O
- One process exits or otherwise terminates
- The OS wants to preempt a process to allow another process to run
- The OS wants to give a newly created process priority
- &c

Context switches can only occur in kernel mode. This is because the lists, queues, and other resources that must be affected during a context switch must alos be protected.

The good news is that any time we might want to perform a context switch, we are already in kernel mode:

- Interrupts signalling a change in an I/O device
- Timer interrupts
- System calls

...each of these events (and many more) cause entry into the kernel mode.

The Idle Process

What happens when there are no user or system processes to run? Perhaps there is no useful work to do,

or all of the useful work is blocked for various reasons.

Most systems run a special process called the idle process.

Rhetorical Question: So, what can the idle process do?
Student answer: Crack MP3.
Instructor Answer: While this might be a useful thing, we actually can't do this. This could be run as low priority process. By defintion, the idle process only runs when nothing else can run.

The semantics of this process are very simple:

x GOTO x

Since the purpose of this process is to occupy an idle CPU, when absolutely nothing else can fill that role, it is critical that this process can never block. And since this process achieves absolutely no useful work, it is critical that it not occupy the CPU when anything else can run.

If we look at the semantics of the process (x GOTO x), it should be fairly easy to convince ourselves that this process can't block.

Student Question: It looks like the only way out of this process is via an interrupt? But there's nothing happening, so one won't occur. And the program doesn't end. How do we get out?

Answer: We don't want to get out until a new process is created and is runnable or an existing process is unblocked. Either requires that an interrupt is generated by an I/O device. The handler for the interrupt gets us out of the idle process and allows the scheduler to run.

Student Question: Can't we perform system maintenance?

Answer: Perhaps, but this cannot replace the idle process, because system maintenance processes can block. And the idle process must not block -- the CPU is running, so it must do something. This type of task can be run with a very low priority, so that the only thing with a lower priority is the idle process. The only other option is to hard-code the scheduler to handle the special case of no runnable processes -- the idle process is a much cleaner fix.

Student Question: Can't we do better? Is there really nothing better?

Answer: No, we can't do anything else -- never mind anything better. This is the universal requirement for the idle process to run. Anything useful might block.

Student Question: What about powering down?

Answer: This is the modern approach. Many systems do reduce power by slowing down the CPU. The idle process still loops -- just more slowly. When something else can run, the CPU speed is restored. Some processors can power down and wake-up when an interrupt occurs.

Student Question: What about polling I/O devices? Don't these want to poll while idle is running?

Answer:

> Modern systems don't actually implement or allow polling. Although a program might poll a device, the program is actually polling the software maintained state of the device. This state is actually maintained using interrupts. Consider the software instruction
>
> ioctl (fd, FIONREAD, &count)
>
> A program can use this to poll a device. it returns the number of characters ready to read. But the infomration accessed by this call is maintained via interrupts. The kernel never busy-waits; this is very inefficient.
>
> There are some very infrequent periodic events that might occur on the order of ever 100mS. But this is very different -- the period is very wide and it isn't really polling.
>
> The other thing is that the idle proces might run 99.9% of the time or it might not run for weeks. There are no guarantees. Polling in the idle process is certainly not safe.

Although the need for the idle process could be obviated by a special case in the scedule, this would mean not only special-case code, but also that the sceduler would need to be interruptable. The idle process is a better solution in the OS, because as with other software, it avoids a nasty special case.

It can be implemented as a regular process with the lowest priority (reserved for idle), or with very simple special case code to ensure that no process is every added behind it in the ready queue.

Student Question: The book talks about process aging. Won't this eventually force the idle process to run, even though other process are ready?

Answer: The books speaks of aging processes. This means raising their priority over time to avoid starvation, regardless of the other facets of the scheduling discipline. If aging is used, the idle process is a special case -- it should not be aged.

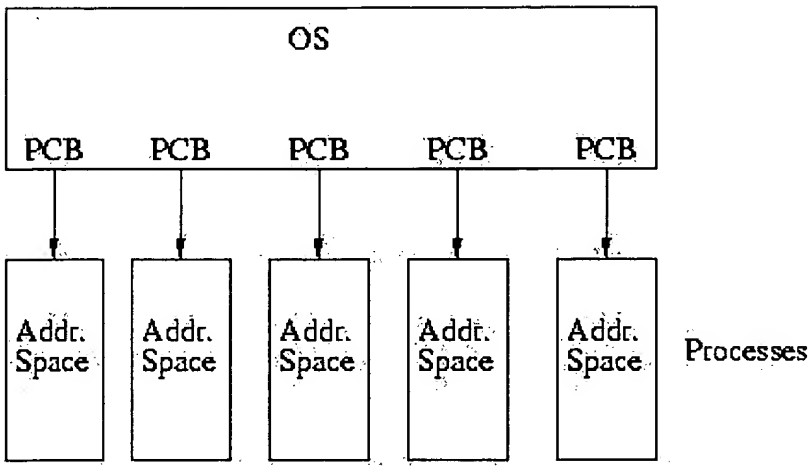
Threads, a.k.a Light-Weight Processes (LWP)

Threads and light-weight processes are, in the context of operating systems, the same thing. Some vendors may assign special meanings to these terms, but this is vendor-specific.

Consider the state of a process: PCB, memory, registers, &c.

Now suppose that we want multiple process-like things that are separately schedulable -- but share memory.

Our model of processes and the OS currently looks like this:



Now consider multiple "processes" sharing the same memory, but otherwise maintaining different state (registers, &c). We call these processes within procesees *threads*. We call them this, because they are separate *threads of control* within the same process (actually, there is a slightly different term for this, but we won't introduce it).

Instructor Question: Why would we want this? Why not just have separate address spaces?

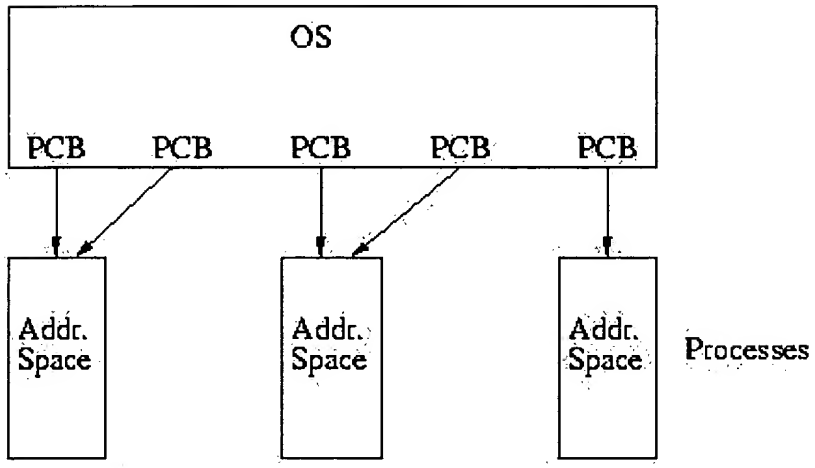
Student Answer: We could have a multiprocessors and this could let us increase parallelism -- multiple threads could run at the same time.

Instructor response: True, but this doesn't justify threads. Multiple processes could run at the same time.

Answer (from class and instructor):

- Fast communication: any thread can write to memory and any other thread can see it -- without penalty.
- Fast context switch: This actually depends on the implementation, but switching among threads within the same address space is faster than switching among processes in different address spaces. We don't need to save and restore the context, including the BASE and LIMIT registers, and other memory-managment registers, to context-switch. But depending on how the threads are implemented, the amount of other overhead can varyy: it can be very, very fast, or just somewhat faster.
- A special cache, called the TLB doesn't need to be flushed to context switch among threads in the same address space. This not only saves the time it takes to flush the cache, but also maintains the utility of the cache -- this is a big win TLB misses are very expensive.
- A process can do useful work, even while it is blocked: yes, but this also depends on the implementation.

Kernel Supported Threads



We can think of kernel supported threads as a system where multiple PCBs can point to the same address space. Each PCB is really no longer a PCB, but more of a *thread control block (TCB)*. But it is still usually called the PCB.

The PCB now maintains the state of each thread and allows each thead to be scheduled independently. The PCB still holds the usual hardware state, queue state, pointer to address state, and a pointer to the *task control block* (TCB).

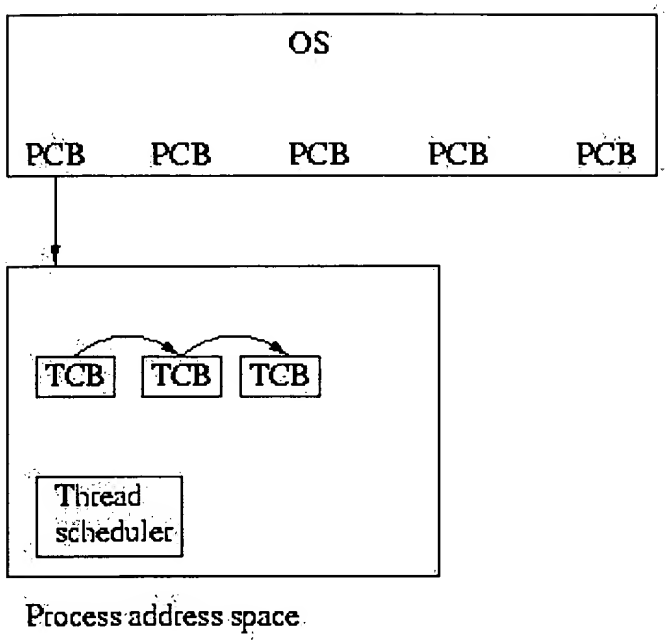
What is a TCB (notice that we are referring to a task control block, **not** a thread control block)? It maintains the state that is not stored in the PCB -- the state that applies to all threads.

We call processes composed of more than one thread of control *tasks*. (This is the term we almost introduced earlier -- it just slipped)

This approach is consistent with the general computer science approach, "anything in computer science can be solved with an additional level of indirection." In this case the indirection via the TCB allows us to have both common and separate state for the threads.

As the name implies, kernel supported threads require OS support. Many, but not all, OSs support kernel threads.

User-level Threads



User-level threads are implemented via a user-level thread library. The kernel niether knows nor cares that they exist. From the kernel's perspective, they are just regular code. User level threads require no changes to the kernel.

TCBs are simply malloc'd each time that a thread is created and linked together to form queues -- within the space of the process.

Context switches among the threads are very cheap -- neither the hardware nor the OS knows or cares about the context of the threads. Since we are not changing address spaces or process state, we are not changing any registers. The state of the threads is just part of the state of the process. This saves much overhead.

Advantages of User-Level Threads

Context switches are cheaper, since no kernel support is required and the state of the threads ispart of the state of the process.

No kernel support is required. user threads can be implemented by user in older OSs that don't support kernel threads, or by more modern, but minimalist OSs that also don't support kernel threads.

User-level threads can implement their own scheduling policy that may be a better fit that the OS process scheduling policy.

Disadvantages of User-Level Threads

If one thread blocks, all threads within the process are blcoked. Example: Reading from the terminal.

Student Question: Can't the threads use non-blocking writes?

Answer: yes, if the programming is careful to use signal-based, non-blocking I/O.

Student Question: How can preemptive scheduling occur without the timer interrupt?

Answer: This can all be simulated within a thread. In this case, one way might be with SIGALRM.

Student Question: Couldn't the threads just yield to each other?

Answer: Yes, this could also work -- since they, unlike processes, are cooperating by design.

Combination: Kernel and User-level Threads?

It is also possible to use a combination of user-level and kernel supported threads. One open research area involves determining exotic ways of balancing the use of both mechanisms.

CPU Scheduling

What are the goals of CPU scheduling? (This also applies to process scheduling and CPU scheduling)

- decrease the overhead of context-switching
- decrease the overhead of the scheduler itself
    - Trade-off: making scheduling decisions more often allows more optimal decisions, becuase the most up-to-date information is used. But running the scheduler frequently increases the overhead of the scheduler itself. This is a tradeoff between scheduling fairness and and decreasing the overhead of the scheduler.
- decrease response time
- increase throughput

Long-Term Scheduling

Long-term schedulign answers the question, "Which jobs are admitted and when?" If too many jobs are scheduled, the system can begin to *thrash* as it spends more time sharing resources than accomplishing useful work. Good long term scheudling ensures that as many processes are run at a time as is possible, without overburdening the system. The goal is to make 100% utilization of resources by interleaving their use among processes -- without making any resource (especially the CPU or memory) too scare to prevent thrashing (or less-sever perfomrance degradation).

This is mostly used in batch systems, since, for example, we would never want login to return "Too many users -- try again later."

Short-Term Scheduling (a.k.a CPU scheduling)

Short term schuling answers the question, which process gets the CPU. This question is particularly important, because the CPU is the most valuable resource in the system.

Medium-Term Scheduling

Medium-term scheduling is often implemented as part of memory management (chapter 8).

Swapping, the movement of a parts of a process's memory to and from disk, to free available RAM, is really medium term scheduling.

If too many processes are consuming too much memory, we keep moving back and forth to disk. It is very expensive to move the same pages to and from disk frequently. It might be better to run fewer processes and keep more of their memory available.

The gola is to give processes the mimimum amount of memory that they need to run efficiently. We can also swap processes to disk when they block waiting for I/O. For example, the user doesn't need CPU while s/he is staring at a prompt wondering what to type.

We'll talk about this topic in detail later.

Back To CPU Scheduling

There are many different types of scheduling. The first major distinction that we will draw is between *preemptive* policies and *non-preemptive* policies.

Premptive approaches allow the scheduler to stop a running process and allow another process to run. With non-preemptive schuduling, once a process starts, it will only be stopped if it voluntarily yields or if it blocks. A stricter definition would require that it remain scheduled, even if it blocked.

We'll start by discussing non-preemptive scheduling:

First Come First Serve (FCFS)

This is the "get in line approach." It is simple and in some sense fair. But long jobs can force short jobs to wait a long time. It may be desirable to run long jobs at night and short jobs during the day, or short jobs first. FCFS can't enforce system priorities.

Shortest Job First(SJF)

SJF scheduling runs the job that requires the least CPU first. How do we know which job this is?

In the past, the programmers on batch systems were required to describe resource utilization on one for the first cards in the deck. If the over specified the CPU use, wasting CPU, they would be penalized by a lower priority.
If they underestimated the CPU use, their job wouldn't finish and they would have to re-run the job. This provided incentive for good estimates.

Now systems might estimate CPU time based on the type of program, the size of the input data set, &c. But these are just guesses. They might be wrong.

This algorithm has the theoretical property that it has the minimal average waiting time across all processes (if the CPU time is estimated exactly). This is because the cost of a long job running first can penalize many small jobs. But if the small jobs run first, only one job is penalized. In other words, the same time penalty can be asmortized over more jobs, if shorter jobs are run first.

But this algorithm is unfair and can lead to starvation. Longer jobs might never run, if shorter jobs keep

arriving.

Priority Scheduling

Priority based schuling is a generalization of either SJF or FCFS. Priority based scheduling runs the highest priority jobs first. If we make shorter jobs a higher priority than larger jobs, it is SJF. If we make older a jobs higher prioirty than newer jobs, it is FCFS.

The priority can be anything. It can be derived from an econimic model. for exmaple, the more you are willing to pay, the sooner your process will get scheduled. This is equivalent to paying for Fed. Ex instead of gorund shipping.

Or perhaps it could be assigned based on other concerns. Perhaps President Cohen 's jobs get a higher priority than ours.

Preemptive Scheduling Is More Interesting

Premptive SJF

A preemptive of SJF would allow the scheduling decision of a process to be re-evaluated based on new jobs that have arrived or based on the quality of the estimate. If a shorter job arived, it could be run. Or if the estimate proved to be inaccurate, and the expected completion time increased, another job might be selected to run.

I/O and CPU bound jobs might be mixed to try to keep all of the I/O devices busy. Estimates could be made of the CPU/IO burst cycle of the processes and they could be scheduled accordingly. These estimates can be adjusted based on past experience using averaging:

Let T_0 be the initial guess
Let C_1 be the initial measured value
T_1, the next guess can be computed as follows:

$T_1 = (T_0 + C_1)/2$

$T_1 = T_0 \text{ alpha} + C_1 (1 - \text{alpha})$

$0 < \text{alpha} < 1$

Subsequent guess can be computed in a similar manner.

This leads to an adaptive evolution of the idea of the CPU/IO burst cycle and an adaptive understanding of the process's priority.

Exhibit F to

Petition for Review by Technology Center SPRE

Microsoft Documentation on Threads

Platform SDK: DLLs, Processes, and Threads

About Processes and Threads

Each *process* provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a base priority, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the *primary thread*, but can create additional threads from any of its threads.

A *thread* is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Microsoft® Windows® supports *preemptive multitasking*, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads.

For more information, see the following topics:

- Multitasking
- Scheduling
- Multiple Threads
- Child Processes

Platform SDK: DLLs, Processes, and Threads

Multiple Threads

A *thread* is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. Each process is started with a single thread, but can create additional threads from any of its threads.

For more information, see the following topics:

- Creating Threads
- Thread Stack Size
- Thread Handles and Identifiers
- Suspending Thread Execution
- Synchronizing Execution of Multiple Threads
- Multiple Threads and GDI Objects
- Thread Local Storage
- Creating Windows in Threads
- Terminating a Thread
- Thread Security and Access Rights

Platform SDK: DLLs, Processes, and Threads

Creating Threads

The **CreateThread** function creates a new thread for a process. The creating thread must specify the starting address of the code that the new thread is to execute. Typically, the starting address is the name of a function defined in the program code (for more information, see **ThreadProc**). This function takes a single parameter and returns a **DWORD** value. A process can have multiple threads simultaneously executing the same function.

The following is a simple example that demonstrates how to create a new thread that executes the locally defined function, ThreadProc. The creating thread uses a dynamically allocated buffer to pass unique information to each instance of the thread function. It is the responsibility of the thread function to free the memory.

The calling thread uses the **WaitForMultipleObjects** function to persist until all worker threads have terminated. Note that if you were to close the handle to a worker thread before it terminated, this does not terminate the worker thread. However, the handle will be unavailable for use in subsequent function calls.

```
#include <windows.h>
#include <strsafe.h>

#define MAX_THREADS 3
#define BUF_SIZE 255

typedef struct _MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

DWORD WINAPI ThreadProc( LPVOID lpParam )
{
    HANDLE hStdout;
    PMYDATA pData;

    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.

    pData = (PMYDATA)lpParam;

    // Print the parameter values using thread-safe functions.

    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n"),
        pData->val1, pData->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars, NULL);

    // Free the memory allocated by the caller for the thread
    // data structure.

    HeapFree(GetProcessHeap(), 0, pData);

    return 0;
}
```

Exhibit G to

Petition for Review by Technology Center SPRE

Hewlett-Packard Documentation of Threads

(Pthread Library)

NAME

pthread - Introduction To POSIX.1c Threads

DESCRIPTION

The POSIX.1c library developed by HP enables the creation of processes that can exploit application and multi-processor platform parallelism. The pthread library *libpthread* consists of over 90 standardized interfaces for developing concurrent applications and synchronizing their actions within processes or between them. This manual page presents an overview of *libpthread* including terminology and how to compile and link programs which use threads.

COMPILATION SUMMARY

A multi-threaded application must define the appropriate POSIX revision level (199506) at compile time and link against the pthread library via -lpthread. For example:

```
cc -D_POSIX_C_SOURCE=199506L -o myapp myapp.c -lpthread
```

All program sources must also include the header file <**pthread.h**>.

Note: When explicitly specifying ANSI compilation (via "-Aa"), defining the POSIX revision level restricts the program to using interfaces within the POSIX namespaces. If interfaces in the larger X/Open namespace are to be called, either of the compiler options, **-D_XOPEN_SOURCE_EXTENDED** or **-D_HPUX_SOURCE**, must be specified in addition to **-D_POSIX_C_SOURCE=199506L**. Alternatively, compiling with -Ae (or not specifying "-A") will implicitly specify **-D_HPUX_SOURCE**.

Note: Some documentation will recommend the use of **-D_REENTRANT** for compilation. While this also functions properly, it is considered an obsolescent form.

THREAD OVERVIEW

A *thread* is an independent flow of control within a process, composed of a context (which includes a register set and a program counter) and a sequence of instructions to execute.

All processes consist of at least one thread. Multi-threaded processes contain several threads. All threads share the common address space allocated for the process. A program using the POSIX pthread APIs creates and manipulates what are called *user threads*. A *kernel thread* is a kernel-schedulable entity which may support one or more user threads. Currently, the HP-UX threads implementation supports only a one-to-one mapping between user and kernel threads.

Each thread is assigned a unique identifier of type *pthread_t* upon creation. The thread id is a process-private value and implementation-dependent. It is considered to be an opaque handle for the thread. Its value should not be used by the application.

p

NOTES ON INTERFACES

The HP-UX system provides some non-standard extensions to the pthread API. These will always have a distinguishing suffix of **_np** or **_NP** (non-portable).

The programmer should always consult the manpages for the functions being used. Some standard-specified functions are not available or may have no effect in some implementations.

THREAD CREATION/DESTRUCTION

A program creates a thread using the **pthread_create()** function. When the thread has completed its work, it may optionally call the **pthread_exit()** function, or simply return from its initial function. A thread can detect the completion of another by using the **pthread_join()** function.

pthread_create() Creates a thread and assigns a unique identifier, *pthread_t*. The caller provides a function which will be executed by the thread. Optionally, the call may explicitly specify some attributes for the thread (see **PTHREAD ATTRIBUTES** below).

pthread_exit() Called by a thread when it completes. This function does not return.

pthread_join() This is analogous to **wait()**, but for pthreads. Any thread may join any other thread in the process, there is no parent/child relationship. It returns when a specified thread terminates, and the thread resources have been reaped.

pthread_detach() Makes it unnecessary to "join" the thread. Thread resources are reaped by the system at the time the thread terminates.

PTHREAD ATTRIBUTES

A set of thread attributes may be provided to **pthread_create()**. Any changes from default values must be made to the attribute set before the call to **pthread_create()** is made. Subsequent changes to the attribute set do not affect the created thread. However, the attribute set may be used in multiple **pthread_create()** calls.

Note that only the "detachstate", "schedparam", "schedpolicy", and "processor" attributes of a thread may be effected subsequent to thread creation. However, this is done via the **pthread_detach()**, **pthread_setschedparam()**, and **pthread_processor_bind_np()** functions, respectively.

pthread_attr_init() Initializes an attribute set for use in the **pthread_create()** call.

pthread_attr_destroy() Destroys the content of an attribute set.

pthread_attr_getdetachstate(),
pthread_attr_getguardsize(),
pthread_attr_getinheritsched(),
pthread_attr_getprocessor_np(),
pthread_attr_getschedparam(),
pthread_attr_getschedpolicy(),
pthread_attr_getscope(),
pthread_attr_getstackaddr(),
pthread_attr_getstacksize(),
pthread_attr_setdetachstate(),
pthread_attr_setguardsize(),
pthread_attr_setinheritsched(),
pthread_attr_setprocessor_np(),
pthread_attr_setschedparam(),
pthread_attr_setschedpolicy(),
pthread_attr_setscope(),
pthread_attr_setstackaddr(),
pthread_attr_setstacksize()

These **pthread_attr_get/set<attribute>()** functions get/set the associated attribute in the attribute set. See the manpages for these functions for descriptions of the attributes.

pthread_default_stacksize_np()
This is used to set the default stacksize for threads created in subsequent attribute set initializations (calls to **pthread_attr_init()**) or in **pthread_create()** where no attributes are supplied.

CANCELLATION

Certain applications may desire to terminate a particular thread without causing the entire process to exit. A thread may be canceled by another thread in the same process while the cancellation target thread executes a system call or particular library routine.

When a thread issues a cancel request against another thread, the target thread can check to see if a request is pending against it via the **pthread_testcancel()** interface. When called with a request pending, the target thread terminates after executing any cleanup handlers which may have been installed. Cleanup handlers may be used to delete any dynamic storage allocated by the canceled thread, to unlock a mutex, or other operations.

Typically, the cancellation type for a thread is *deferred*. That is, cancellation requests are held pending until the thread reaches a *cancellation*point which is simply one of a list of library functions and system calls (see lists below).

The thread may set its cancellation type to *asynchronous*. In this case cancellation requests are acted upon at any time. This can be used effectively in compute-bound threads which do not call any functions that are cancellation points.

pthread_cancel() Cancel execution of a given thread.

pthread_testcancel() Called by a thread to process pending cancel requests.

pthread_setcancelstate(),
pthread_setcanceltype()

Set the characteristics of cancellation for the thread. Cancellation may be enabled or disabled, or it may be synchronous or deferred.

pthread_cleanup_pop(),
pthread_cleanup_push()

Register or remove cancellation cleanup handlers.

Cancellation points in the *pthread* library:

pthread_testcancel()
pthread_cond_wait()
pthread_cond_timedwait()
pthread_join()

System functions which are always cancellation points:

accept()	aio_suspend()	close()	connect()
creat()	dup2()	*fcntl()	fsync()
getdirentries()	getmsg()	getpmsg()	ioctl()
lockf()	lockf64()	lseek()	lseek64()
mq_receive()	mq_send()	msgrcv()	msgsnd()
msync()	nanosleep()	open()	pause()
poll()	putmsg()	putpmsg()	read()
readv()	recv()	recvfrom()	recvmsg()
rename()	select()	semop()	send()
sendmsg()	sendto()	sigsuspend()	sigtimedwait()
sigwait()	sigwaitinfo()	socket()	system()
wait()	wait3()	waitid()	waitpid()
write()	writev()		

* **fcntl()** is a cancellation point only with the **F_SETLK** command.

For the following *libc* functions, whether the thread is cancelled depends upon what action is performed while executing the function. If the thread blocks while inside the function, a cancellation point is created (i.e., the thread may be cancelled). Note: Other libraries may have cancellation points. Check the associated documentation for details.

blclose()	blget()	blopen()
blread()	blset()	catclose()
catgets()	catopen()	closedir()
closelog()	creat64()	ctermid()
cuserid()	dbm_close()	dbm_delete()
dbm_fetch()	dbm_firstkey()	dbm_nextkey()
dbm_open()	dbm_store()	dbmclose()
endexportent()	endfsent()	endgrent()
endhostent()	endnetent()	endnetgrent()
endprotoent()	endpwent()	endservent()
endutxent()	fclose()	fflush()
fgetc()	fgetgrent()	fgetpos()
fgetpos64()	fgetpwent()	fgets()
fgetwc()	fgetws()	fmtmsg()
fopen()	fopen64()	fprintf()
fputc()	fputs()	fputwc()
fputws()	fread()	freopen()
freopen64()	fsetpos()	fsetpos64()
ftell()	ftello()	ftello64()
ftw()	ftw64()	fwrite()
getc()	getc_unlocked()	getchar()
getchar_unlocked()	getcwd()	getdate()
getgrent()	getgrgid()	getgrgid_r()
getgrnam()	getgrnam_r()	gethostbyaddr()
gethostbyname()	gethostent()	getlogin()
getlogin_r()	getopt()	getpass()

p

getpw()	getpwent()	getpwnam()
getpwnam_r()	getpwuid()	getpwuid_r()
gets()	getservbyname()	getservbyport()
getservent()	gettxt()	getusershell()
getutent()	getutid()	getutline()
getutxent()	getutxid()	getutxline()
getw()	getwc()	getwchr()
getwd()	glob()	grantpt()
iconv_close()	iconv_open()	initgroups()
lckpwdf()	mkstemp()	msem_lock()
nftw()	nftw2()	nftw64()
nlist()	open64()	opendir()
openlog()	pclose()	perror()
pfmt()	popen()	prealloc()
prealloc64()	printf()	putc()
putc_unlocked()	putchar()	putchar_unlocked()
putpwent()	puts()	pututline()
pututxline()	putw()	putwc()
putwchar()	putws()	readdir()
readdir_r()	realpath()	remove()
rewind()	rewinddir()	scandir()
scanf()	seekdir()	setgrent()
sethostent()	setnetent()	setnetgrent()
setprotoent()	setpwent()	setservent()
setusershell()	setutent()	setutxent()
sigpause()	sleep()	strerror()
syslog()	tcdrain()	tell()
tmpfile()	tmpfile64()	tmpnam()
ttyname()	ttyname_r()	ttyslot()
ulckpwdf()	ungetc()	ungetwc()
usleep()	vfprintf()	vfscanf()
vprintf()	vscanf()	vsprintf()
vsscanf()	wordexp()	wordfree()

NOTE1: The above functions may not be fully supported or may be considered obsolete. Consult individual manpages for more info.

NOTE2: The list of cancellation points will vary from release to release. In general, if a function can return with an **EINTR** error, chances are that it is a cancellation point.

SCHEDULING

Threads may individually control their scheduling policy and priorities. Threads may also suspend their own execution, or that of other threads. Finally, threads are given some control over allocation of processor resources.

pthread_suspend() This function is used to temporarily stop the execution of a thread.

pthread_continue(),
pthread_resume_np()

These functions cause a previously suspended thread to continue execution.

pthread_num_processor_np(),
pthread_processor_bind_np(),
pthread_processor_id_np()

These functions are used to interrogate processor configuration and to bind a thread to a specific processor.

pthread_getconcurrency(),
pthread_setconcurrency()

These functions are used to control the actual concurrency for unbound threads.

pthread_getschedparam(),
pthread_setschedparam()

These functions are used to manipulate the scheduling policy and priority for a thread.

sched_get_priority_max(),
sched_get_priority_min()

These functions are used to interrogate the priority range for a given scheduling policy.

sched_yield This function is used by a thread to yield the processor to other threads of equal or greater priority.

COMMUNICATION & SYNCHRONIZATION

Multi-threaded applications concurrently execute instructions. Access to process-wide (or interprocess) shared resources (memory, file descriptors, etc.) requires mechanisms for coordination or synchronization among threads. The *libpthread* library offers synchronization primitives necessary to create a deterministic application. A multi-threaded application ensures determinism by forcing asynchronous thread contexts to synchronize, or serialize, access to data structures and resources managed and manipulated during run-time. These are mutual-exclusion (mutex) locks, condition variables, and read-write locks. The HP-UX operating system also provides POSIX semaphores (see next section).

Mutexes furnish the means to exclusively guard data structures from concurrent modification. Their protocol precludes more than one thread which has locked the mutex from changing the contents of the protected structure until the locker performs an analogous mutex unlock. A mutex can be initialized in two ways: by a call to **pthread_mutex_init()**; or by assignment of PTHREAD_MUTEX_INITIALIZER.

Condition Variables are used by a thread to wait for the occurrence of some event. A thread detecting or causing such an event can *signal* or *broadcast* that occurrence to the waiting thread or threads.

Read-Write locks permit concurrent read access by multiple threads to structures guarded by a read-write lock, but write access by only a single thread.

pthread_mutex_init(),
pthread_mutex_destroy()

Initialize/destroy contents of a mutex lock.

pthread_mutex_lock(),
pthread_mutex_trylock(),
pthread_mutex_unlock()

Lock/unlock a mutex.

pthread_mutex_getprioceiling(),
pthread_mutex_setprioceiling()

Manipulate mutex locking priorities.

pthread_mutexattr_init(),
pthread_mutexattr_destroy(),
pthread_mutexattr_getprioceiling(),
pthread_mutexattr_getprotocol(),
pthread_mutexattr_getpshared(),
pthread_mutexattr_gettype(),
pthread_mutexattr_getspin_np(),
pthread_mutexattr_setprioceiling(),
pthread_mutexattr_setprotocol(),
pthread_mutexattr_setpshared(),
pthread_mutexattr_settype(),
pthread_mutexattr_setspin_np()

Manage mutex attributes used for **pthread_mutex_init()**. Only the "prioceiling" attribute can be changed for an exiting mutex.

pthread_mutex_getyieldfreq_np(),
pthread_mutex_setyieldfreq_np()

These functions, together with the *spin* attributes, are used to tune mutex performance to the specific application.

pthread_cond_init(),
pthread_cond_destroy()

Initialize/destroy contents of a read-write lock.

pthread_cond_signal(),
pthread_cond_broadcast(),
pthread_cond_timedwait(),
pthread_cond_wait()

Wait upon or signal occurrence of a condition variable.

pthread_condattr_init(),
pthread_condattr_destroy(),
pthread_condattr_getpshared(),
pthread_condattr_setpshared()

Manage condition variable attributes used for **pthread_cond_init()**.

pthread_rwlock_init(),
pthread_rwlock_destroy()

Initialize/destroy contents of a read-write lock.

pthread_rwlock_rdlock(),
pthread_rwlock_tryrdlock(),
pthread_rwlock_wrlock(),
pthread_rwlock_trywrlock(),
pthread_rwlock_unlock()

Lock/unlock a read-write lock.

pthread_rwlockattr_init(),
pthread_rwlockattr_destroy(),
pthread_rwlockattr_getpshared(),
pthread_rwlockattr_setpshared()

Manage read-write lock attributes used for **pthread_rwlock_init()**.

POSIX 1.b SEMAPHORES

The semaphore functions specified in the POSIX 1.b standard can also be used for synchronization in a multithreaded application.

sem_init(),
sem_destroy()

Initialize/destroy contents of a semaphore.

sem_post(),
sem_wait(),
sem_trywait()

Increment/decrement semaphore value (possibly blocking).

SIGNALS

In a multithreaded process, all threads share signal actions. That is, a signal handler established by one thread is used in all threads. However, each thread has a separate signal mask, by which it can selectively block signals.

Signals can be sent to other threads within the same process, or to other processes. When a signal is sent to the process, exactly one thread which does not have that signal blocked will handle the signal. When sent to a thread within the same process, that thread will handle the signal, perhaps later if the signal is blocked. Signals whose action is to terminate, stop, or continue will terminate, stop, or continue the entire process, respectively, even if directed at a particular thread.

pthread_kill() Sends a signal to the given thread.

pthread_sigmask() Blocks selected signals for the thread.

sigwait(),
sigwaitinfo(),
sigtimedwait() These functions synchronously wait for given signals.

THREAD-SPECIFIC DATA

Thread-specific data (TSD) is global data that is private or specific to a thread. Each thread has a different value for the same thread-specific data variable. The global *errno* is a perfect example of thread-specific global data.

Each thread-specific data item is associated with a key. The key is shared by all threads. However, when a thread references the key, it references its own private copy of the data.

pthread_key_create(),
pthread_key_destroy()

These functions manage the thread-specific data keys.

pthread_getspecific(),
pthread_setspecific()

These functions retrieve and assign the data value associated with a key.

The HP-UX compiler supports a *thread local storage* (TLS) storage class. (This is not a POSIX standard feature.) TLS is identical to TSD, except functions are not required to create/set/get values. TLS variables are accessed just like normal global variables. TLS variables can be declared using the following syntax:

```
__thread      int      zyx;
```

The keyword **__thread** tells the compiler that **zyx** is a TLS variable. Now each thread can set or get TLS with statements such as:

```
zyx = 21;
```

Each thread will have a different value associated with **zyx**.

TLS variables cannot be statically initialized, all are initially zero. Dynamically loaded libraries (via **shl_load()**) cannot declare (but may use) TLS variables.

TLS does have a cost in thread creation/termination operations, as TLS space for each thread must be allocated and zeroed, regardless of whether it ever will use the variables. If few threads actually use a large TLS area, it may be wise to instead use the POSIX TSD (above).

REENTRANT LIBC & STDIO

Because they return pointers to library-internal static data, a number of *libc* functions cannot be used in multi-threaded programs. This is because calling these functions in a thread will overwrite the results of previous calls in other threads. Alternate functions, having the suffix **_r** (for reentrant), are provided within *libc* for threaded programming.

Also, some primitives for synchronization of standard I/O operations are provided.

asctime_r(),
ctime_r(),
getgrgid_t(),
getgrnam_r(),
getlogin_r(),
getpwnam_r(),
getpwuid_r(),
gmtime_t(),
localtime_r(),
rand_r(),
readdir_r(),
strtok_r(),
ttyname_r()

Provide reentrant versions of previously existing *libc* functions.

flockfile(),
ftrylockfile(),
funlock()

Provide explicit synchronization for standard I/O streams.

MISCELLANEOUS FUNCTIONS

The section summarizes some miscellaneous pthread-related functions not covered in the preceding sections.

pthread_atfork()	Establish special functions to be called just prior to and just subsequent to a **fork()** operation.
pthread_equal()	Tests whether two pthread_t values represent the same pthread.
pthread_once()	Executes given function just once in a process, regardless of how many threads make the same call. (Useful for one-time data initialization.)
pthread_self()	Returns identifier (pthread_t) of calling thread.

THREAD DEBUGGING

Debugging of multithreaded programs is supported in the standard HP-UX debugger, **dde**. When any thread is to be stopped due to a debugger event, the debugger will stop all threads. The register state, stack, and data for any thread can be interrogated and manipulated.

See the *dde*(1) manpage and built-in graphical help system for more information.

TRACING FACILITIES

HP-UX provides a tracing facility for pthread operations. To use it, you must link your application using the tracing version of the library:

```
cc -D_POSIX_C_SOURCE=199506L -o myapp myapp.c -lpthread_tr -lcl
```

When the application is executed, it produces a per-thread file of pthread events. This is used as input to the **ttv** thread trace visualizer facility available in the **HP/PAK performance application kit**.

There are environment variables defined to control trace data files:

THR_TRACE_DIR
Where to place the trace data files. If this is not defined, the files go to the current working directory.

THR_TRACE_SYNC
By default, trace records are buffered and only written to the file when the buffer is full. If this variable is set to any non-NULL value, data is immediately written to the trace file.

THR_TRACE_EVENTS
By default, all pthread events are traced. If this variable is defined, only the categories defined will be traced. Each category is separated by a ':'. The possible trace categories are:

```
thread:cond:mutex:rwlock
```

For example, to only trace thread and mutex operations set the THR_TRACE_EVENTS variable to:

```
thread:mutex
```

Details of the trace file record format can be found in **/usr/include/sys/trace_thread.h**.

See the *ttv*(1) manpage and built-in graphical help system for more information on the use of the trace information.

PERFORMANCE CONSIDERATIONS

Often, an application is designed to be multithreaded to improve performance over its single-threaded counterparts. However, the multithreaded approach requires some attention to issues not always of concern in the single-threaded case. These are issues traditionally associated with the programming of multiprocessor systems.

The design must employ a *lock granularity* appropriate to the data structures and access patterns. *Coarse-grained* locks, which protect relatively large amounts of data, can lead to undesired lock *contention*, reducing the potential parallelism of the application. On the other hand, employing very *fine-grained* locks, which protect very small amounts of data, can consume processor cycles with too much locking activity.

The use of *thread-specific data* (TSD) or *thread-local*storage (TLS) must be traded off, as described above (see **THREAD-SPECIFIC DATA**).

Mutex *spin* and *yield frequency* attributes can be used to tune mutex behavior to the application. See *pthread_mutexattr_setspin_np*(3T) and *pthread_mutex_setyieldfreq_np*(3T) for more information.

The *default stacksize* attribute can be set to improve system thread caching behavior. See *pthread_default_stacksize_np*(3T) for more information.

Because multiple threads are actually running simultaneously, they can be accessing the same data from multiple processors. The hardware processors coordinate their caching of data such that no processor is

using *stale data*. When one processor accesses the data (especially for write operations), the other processors must flush the stale data from their caches. If multiple processors repeatedly read/write the same data, this can lead to *cache-thrashing* which slows execution of the instruction stream. This can also occur when threads access separate data items which just happen to reside in the same hardware-cachable unit (called a *cache line*). This latter situation is called *false-sharing* which can be avoided by spacing data such that popular items are not stored close together.

GLOSSARY

The following definitions were extracted from the text *ThreadTime* by Scott J. Norton and Mark D. DiPasquale, Prentice-Hall, ISBN 0-13-190067-6, 1996.

Application Programming Interface (API)
An interface is the conduit that provides access to an entity or communication between entities. In the programming world, an interface describes how access (or communication) with a function should take place. Specifically, the number of parameters, their names and purpose describe how to access a function. An API is the facility that provides access to a function.

Async-Cancel Safe
A function that may be called by a thread with the cancelability state set to **PTHREAD_CANCEL_ENABLE** and the cancelability type set to **PTHREAD_CANCEL_ASYNCHRONOUS**. If a thread is canceled in one of these functions, no state is left in the function. These functions generally do not acquire resources to perform the function's task.

Async-Signal Safe
An async-signal safe function is a function that may be called by a signal handler. Only a restricted set of functions may safely be called by a signal handler. These functions are listed in section 3.3.1.3 of the POSIX.1c standard.

Asynchronous Signal
An asynchronous signal is a signal that has been generated due to an external event. Signals sent via **kill()** and signals generated due to timer expiration or asynchronous I/O completion are all examples of asynchronously generated signals. Asynchronous signals are delivered to the process. All signals can be generated asynchronously.

Atfork Handler
Application-provided and registered functions that are called before and after a **fork()** operation. These functions generally acquire all mutex locks before the **fork()** and release these mutex locks in both the parent and child processes after the **fork()**.

Atomic Operation
An operation or sequence of events that is guaranteed to complete as if it were one instruction.

Barrier
A synchronization primitive that causes a certain number of threads to wait or rendezvous at specified points in an application. Barriers are used when a application needs to ensure that all threads have completed some operation before proceeding onto the next task.

Bound Thread
A user thread that is directly bound to a kernel-scheduled entity. These threads contain a system scheduling scope and are scheduled directly by the kernel.

Cache Thrashing
Cache thrashing is a situation in which a thread executes on different processors, causing cached data to be moved to and from the different processor caches. Cache thrashing can cause severe performance degradation.

Cancellation Cleanup Handler
An application-provided and registered function that is called when a thread is canceled. These functions generally perform thread cleanup actions during thread cancellation. These handlers are similar to signal handlers.

Condition Variable
A condition variable is a synchronization primitive used to allow a thread to wait for an event. Condition variables are often used in producer-consumer problems where a producer must provide something to one or more consumers.

Context Switch
The act of removing the currently running thread from the processor and running another thread. A context switch saves the register state of the currently running thread and restores the register state of the thread chosen to execute next.

Critical Section
A section of code that must complete atomically and uninterrupted. A critical section of code is generally one in which some global resource (variables, data structures, linked lists, etc.) is modified. The operation being performed must complete atomically so that other threads do not see the critical section in an inconsistent state.

Deadlock
A deadlock occurs when one or more threads can no longer execute. For example, thread *A* holds lock 1 and is blocked on lock 2. Meanwhile, thread *B* holds lock 2 and is blocked on lock 1. Threads *A* and *B* are permanently deadlocked. Deadlocks can occur with any number of resource holding threads. An *interactive deadlock* involves two or more threads. A *recursive* (or *self) deadlock* involves only one thread.

Detached Thread
A thread whose resources are automatically released by the system when the thread terminates. A detached thread cannot be joined by another thread. Consequently, detached threads cannot return an exit status.

Joinable Thread
A thread whose termination can be waited for by another thread. Joinable threads can return an exit status to a joining thread. Joinable threads maintain some state after termination until they are joined by another thread.

Kernel Mode
A mode of operation where all operations are allowed. While a thread is executing a system call it is executing in kernel mode.

Kernel Space
The kernel program exists in this space. Kernel code is executed in this space at the highest privilege level. In general, there are two privilege levels: one for user code (user mode) and the other for kernel code (kernel mode).

Kernel Stack
When a thread makes a system call, it executes in kernel mode. While in kernel mode, it does not use the stack allocated for use by the application. Instead, a separate kernel stack is used while in the system call. Each kernel-scheduled entity, whether a process, kernel thread or lightweight process, contains a kernel stack. See *Stack* for a generic description of a stack.

Kernel Thread
Kernel threads are created by the thread functions in the threads library. Kernel threads are *kernel-scheduled entities* that are visible to the operating system kernel. A kernel thread typically supports one or more user threads. Kernel threads execute kernel code or system calls on behalf of user threads. Some systems may call the equivalent of a kernel thread a *lightweight process.* See *Thread* for a generic description of a thread.

Lightweight Process
A kernel-scheduled entity. Some systems may call the equivalent of a lightweight process a kernel thread. Each process contains one or more lightweight process. How many lightweight processes a process contains depends on whether and how the process is multithreaded. See *Thread* for a generic description of a thread.

Multiprocessor
A system with two or more processors (CPUs). Multiprocessors allow multithreaded applications to obtain true parallelism.

Multithreading
A programming model that allows an application to have multiple threads of execution. Multithreading allows an application to have concurrency and parallelism (on multiprocessor systems).

Mutex
A mutex is a mutual exclusion synchronization primitive. Mutexes provide threads with the ability to regulate or serialize access to process shared data and resources. When a thread locks a mutex, other threads trying to lock the mutex block until the owning thread unlocks the mutex.

POSIX
Portable Operating System Interface. POSIX defines a set of standards that multiple vendors conform to in order to provide for application portability. The Pthreads standard (POSIX 1003.1c) provides a set of portable multithreading APIs to application developers.

Priority Inversion
A situation where a low-priority thread has acquired a resource that is needed by a higher priority thread. As the resource cannot be acquired, the higher priority thread must wait for the resource. The end result is that a low-priority thread blocks a high-priority thread.

Process
A process can be thought of as a container for one or more threads of execution, an address space, and shared process resources. All processes have at least one thread. Each thread in the process executes within the process' address space. Examples of process-shared resources are open file descriptors, message queue descriptors, mutexes, and semaphores.

Process Control Block (PCB)
This structure holds the register context of a process.

Process Structure
The operating system maintains a process structure for each process in the system. This structure represents the actual process internally in the system. A sample of process structure information includes the process ID, the process' set of open files, and the signal vector. The process structure and the values contained within it are part of the context of a process.

Program Counter (PC)
The program counter is part of the register context of a process. It holds the address of the current instruction to be executed.

Race Condition
When the result of two or more threads performing an operation depends on unpredictable timing factors, this is a race condition.

Read-Write Lock
A read-write lock is a synchronization primitive. Read-write locks provide threads with the ability to regulate or serialize access to process-shared data and resources. Read-write locks allow multiple readers to concurrently acquire the read lock whereas only one writer at a time may acquire the write lock. These locks are useful for shared data that is mostly read and only rarely written.

Reentrant Function
A reentrant function is one that when called by multiple threads, behaves as if the function was called serially, one after another, by the different threads. These functions may execute in parallel.

Scheduling Allocation Domain
The set of processors on which a thread is scheduled. The size of this domain may dynamically change over time. Threads may also be moved from one domain to another.

Scheduling Contention Scope
The scheduling contention scope defines the group of threads that a thread competes with for access to resources. The contention scope is most often associated with access to a processor. However, this scope may also be used when threads compete for other resources. Threads with the system scope compete for access to resources with all other threads in the system. Threads with the process scope compete for access to resources with other process scope threads in the process.

Scheduling Policy
A scheduling policy is a set of rules used to determine how and when multiple threads are scheduled to execute. The scheduling policy also determines how long a thread is allowed to execute.

Scheduling Priority
A scheduling priority is a numeric priority value assigned to threads in certain scheduling policies. Threads with higher priorities are given preference when scheduling decisions are made.

Semaphore
A semaphore is similar to a mutex. A semaphore regulates access to one or more shared objects. A semaphore has a value associated with it. The value is generally set to the number of shared resources regulated by the semaphore. When a semaphore has a value of one, it is a binary semaphore. A mutex is essentially a binary semaphore. When a semaphore has a value greater than one, it is known as a *counting semaphore*. A counting semaphore can be locked by multiple threads simultaneously. Each time the semaphore is locked, the value is decremented by one. After the value reaches zero, new attempts to lock the semaphore cause the locking thread to block until the semaphore is unlocked by another thread.

Shared Object
A shared object is a tangible entity that exists in the address space of a process and is accessible by all threads within the process. In the context of multithreaded programming, "shared objects" are global variables, file descriptors, and other such objects that require access by threads to be synchronized.

Signal
A signal is a simplified IPC mechanism that allows a process or thread to be notified of an event. Signals can be generated synchronously and asynchronously.

Signal Mask
A signal mask determines which signals a thread accepts and which ones are blocked from delivery. If a synchronous signal is blocked from delivery, it is held pending until either the thread unblocks the signal or the thread terminates. If an asynchronous signal delivered to the process is blocked from delivery by a thread, the signal may be handled by a different thread in the process that does not have the signal blocked.

Signal Vector
A signal vector is a table contained in each process that describes the action that should be taken when a signal is delivered to a thread within the process. Each signal has one of three potential behaviors: ignore the signal, execute a signal-handling function, or perform the default action of the signal (usually process termination).

Single-Threaded
means that there is only one *flow of control* (one thread) through the program code; only one instruction is executed at a time.

Spinlock
A synchronization primitive similar to a mutex. If the lock cannot be acquired, instead of blocking, the thread wishing to acquire the lock spins in a loop until the lock can be acquired. Spinlocks can be easily used improperly and can severely degrade performance if used on a single processor system.

Spurious Wakeup
A spurious wakeup occurs when a thread is incorrectly unblocked, even though the event it was waiting for has not occurred. A condition wait that is interrupted and returns because the blocked thread received a normal signal is an example of a spurious wakeup.

Stack
A stack is used by a thread to make function calls (and return from those calls), to pass arguments to a function call, and to create the space for local variables when in that function call. Bound threads have a user stack and a kernel stack. Unbound threads have only a user stack.

Synchronous Signal
A synchronous signal is a signal that has been generated due to some action of a specific thread. For example, when a thread does a divide by zero, causes a floating point exception, or executes an illegal instruction, a signal is generated synchronously. Synchronous signals are delivered to the thread that caused the signal to be sent.

Traditional Process
This is a single-threaded entity that can be scheduled to execute on a processor.

Thread
A thread is an independent flow of control within a process, composed of a context (which includes a register set and program counter) and a sequence of instructions to execute.

Thread Local Storage (TLS)
Thread local storage is essentially thread-specific data requiring support from the compilers. With TLS, an application can allocate the actual data as thread-specific data rather than using thread-specific data keys. Additionally, TLS does not require the thread to make a function call to obtain thread-specific data. The thread can access the data directly.

Thread-Safe Function
A thread-safe function is one that may be safely called by multiple threads at the same time. If the function accesses shared data or resources, this access is regulated by a mutex or some other form of synchronization.

Thread-Specific Data (TSD)
Thread-specific data is global data that is specific to a thread. All threads access the same data variable. However, each thread has its own thread-specific value associated with this variable. errno is an example of thread-specific data.

Thread Structure
The operating system maintains a thread structure for each thread in the system. This structure represents the actual thread internally in the system. A sample of thread structure information includes the thread ID, the scheduling policy and priority, and the signal mask. The thread structure and the values contained within it are part of the context of a thread.

User Mode
A mode of operation where a subset of operations are allowed. While a thread is executing an applications code, it is executing in user mode. When the thread makes a system call, it changes modes and executes in kernel mode until the system call completes.

User Space
The user code exists in this space. User code is executed in this space at the normal privilege level. In general, there are two privilege levels: one for user code (user mode) and the other for kernel code (kernel mode).

User Stack
When a thread is executing code in user space, it needs to use a stack to make function calls, pass parameters, and create local variables. While in user mode, a thread does not use the kernel stack. Instead, a separate user stack is allocated for use by each user thread. See *Stack* for a generic description of a stack.

User Thread
When **`pthread_create()`** is called, a user thread is created. Whether a kernel-scheduled entity (kernel thread or lightweight process) is also created depends on the user thread's scheduling contention scope. When a bound thread is created, both a user thread and a kernel-scheduled entity are created. When an unbound thread is created, generally only a user thread is created. See *Thread* for a generic description of a thread.

SEE ALSO

ThreadTime by Scott J. Norton and Mark D. DiPasquale, Prentice-Hall, ISBN 0-13-190067-6, 1996.

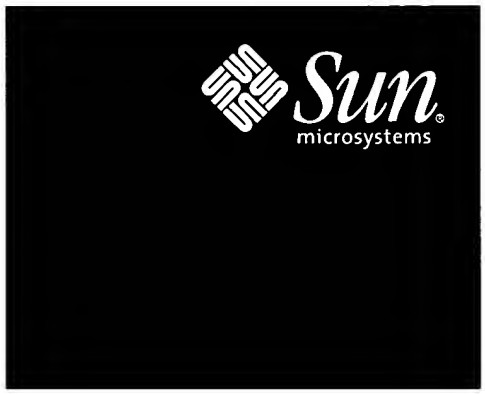
Exhibit H to

Petition for Review by Technology Center SPRE

Sun Microsystems Documentation for Threads

Multithreading in the Solaris™ Operating Environment

A Technical White Paper



Chapter 1

Introduction

Multithreading is a popular programming and execution model that allows multiple threads to exist within the context of a single process, sharing the process' resources but able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. However, perhaps the most interesting application of the technology is when it is applied to a single process to enable parallel execution on a multiprocessor system.

Multithreading is not new to the Solaris™ Operating Environment (OE). Threads have played an important part in enabling Sun to deliver successful, scalable, multiprocessor systems. The threading capabilities in the Solaris OE are continually being improved, and the Solaris 9 Operating Environment contains innovations for multithreaded applications — chiefly, the adoption of a highly tuned and tested "1:1" thread model in preference to the historic "MxN" implementation.

The goal of this paper is to describe the new threads implementation and encourage its use. It begins by providing historical context and recent developments. A technical audience with practical experience of multithreading in the C or Java™ programming languages or similar environments is assumed. For additional reading on the subject, *"Programming with Threads"* is highly recommended for novice and expert alike.

A Decade of Change

Many changes occurred during the first ten years that Sun provided support for multithreaded applications:

- System size (in terms of CPUs and memory) saw exponential growth
- Multithreaded programming techniques evolved and matured
- Application programming interfaces for threads became standardized

- Threads became ubiquitous in modern operating systems
- Use of the inherently threaded Java programming language became widespread
- Threaded applications became the norm, not the exception

Clearly, the most significant development has been moving multithreading from theory into practice. Along the way, the multithreaded computing landscape has changed. Factors which seemed important 10 years ago may be irrelevant today, and issues which were once considered on the fringe are now commanding the full attention of developers.

Responding to Change

The Solaris OE has always moved with the times, and a great deal has been done to improve the scalability of threads within the kernel. This has led to exciting, and sometimes surprising, innovations. One such innovation is the move away from the original MxN model to a 1:1 implementation.

Simply a Better Implementation

The Solaris 8 OE introduced an alternate implementation based on the 1:1 model which leveraged the improvements in kernel threading directly at the process level. The 1:1 implementation was simpler to develop, and its code paths were generally more efficient than those of the old implementation. In the Solaris 8 OE, the MxN implementation remains the default. Users have tested the new 1:1 implementation, and many have adopted it. Again, this is not to say that a good implementation of the MxN model is impossible, but simply that a good 1:1 implementation is probably sufficient.

Note – This paper does not attempt a discussion of the relative merits of the MxN and 1:1 threading models. The basic thesis is that the quality of an implementation is often more important.

Scalable Threaded Applications

New programming technologies often go through an early period of upheaval as developer communities begin to separate theory from practice. Initial naivete is tempered by realism and experience. Witness the history of the Java™ programming language: the initial focus on browsers and not-so-thin-client applications has matured, broadened, and shifted to embrace just about every level of computing — from consumer products to the data center. Multithreading has seen a similar evolution, and one of the shifts in emphasis has been away from "threads for everything" to "threads for scalability."

For example, developers used to believe that a threads stack was an ideal place to store application session state. However, experience has proved that it is generally better to hold session state in well-designed data structures, and deploy a worker thread pool to deliver scalable, predictable performance.

Exhibit I to

Petition for Review by Technology Center SPRE

Digital Equipment Corp. Documentation for OpenVMS DECthreads

HP OpenVMS Systems Documentation

Guide to DECthreads

Order Number: AA--QSBPC--TE

January 1999

This guide reviews the principles of multithreaded programming, as reflected in the IEEE POSIX 1003.1c-1995 standard, provides implementation guidelines and reference information for DECthreads, Compaq's Multithreading Run-Time Libra

Revision/Update Information: This manual supersedes the *Guide to DECthreads*, Version 7.1.

Software Version: OpenVMS Alpha Version 7.2
OpenVMS VAX Version 7.2

Compaq Computer Corporation
Houston, Texas

The OpenVMS documentation set is available on CD-ROM.

This document was prepared using VAX DOCUMENT, Version V3.2n.

Contents | Index

Preface
Intended Audience

This guide is for system and application programmers who use DECthreads to create multithreaded applications or to cr safe code libraries that can be called from single-threaded or multithreaded applications.
Document Structure

This guide consists of the following:

Part 1

- Chapter 1 provides a brief overview of multithreaded programming.
- Chapter 2 discusses the concepts and techniques related to DECthreads.
- Chapter 3 describes thread disciplines and coding issues you may face when writing a multithreaded program.
- Chapter 4 addresses writing thread-safe libraries.
- Chapter 5 introduces and provides conventions for the modular use of the DECthreads exception package.
- Chapter 6 contains an example demonstrating how to call DECthreads routines from a C language program.

Part 2

- This part provides detailed reference information on each **pthread** interface routine. Routine descriptions appear alphabetical order by routine name.

Part 3

- This part provides detailed reference information on each **tis** interface routine. Routine descriptions appear in alp order by routine name.

Part 4 - Appendixes

- Appendix A discusses DECthreads issues and restrictions specific to Compaq's DIGITAL UNIX systems.
- Appendix B discusses DECthreads issues and restrictions specific to OpenVMS systems.
- Appendix C discusses DECthreads issues and restrictions specific to the Microsoft Win32 interfaces on Windows systems.
- Appendix D discusses debugging issues for a multithreaded program that uses DECthreads.
- Appendix E summarizes the differences between the Compaq proprietary DECthreads CMA (or **cma**) interface a

DECthreads **pthread** interface. Use this appendix to help you migrate your programs and applications to the **pthr** interface.

- Appendix F summarizes the differences between the DECthreads POSIX 1003.4a/Draft 4 (or **d4**) interface and th DECthreads **pthread** interface. Use this appendix to help you migrate your programs and applications to the **pthr** interface.

Glossary

- The Glossary contains definitions of terms used in this guide, listed alphabetically.

Related Documents

For additional information on the Open Systems Software Group (OSSG) products and services, access the following Or World Wide Web address:

```
http://www.openvms.digital.com
```

Reader's Comments

Compaq welcomes your comments on this manual.

Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**writer@dceidl.enet.dec.com**
Fax	603 884-0120, Attention: Core Technology Group, ZKO2-3/Q18
Mail	Core Technology Group, ZKO2-3/Q18 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following World Wide Web address to order additional documentation:

```
http://www.openvms.digital.com:81/
```

If you need help deciding which documentation best meets your needs, call (800-282-6672).

Conventions

VMScluster systems are now referred to as OpenVMS Cluster systems. Unless otherwise specified, references to Open\ Clusters or clusters in this document are synonymous with VMSclusters.

The following conventions are used in this manual:

...	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the iter omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentl you choose more than one.
[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file

	specification or in the syntax of a substring specification in an assignment statement.)
[|]	In command format descriptions, vertical bars separating items inside brackets indicate that you ch none, or more than one of the options.
{ | }	In command format descriptions, braces indicate required elements; you must choose one of the o listed.
text style	This text style represents the introduction of a new term or the name of an argument, an attribute, reason. In the HTML version of this document, this convention appears as *italic text*.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables includ information that varies in system output (Internal error *number*), in command lines (/PRODUCER= and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keyword names of independently compiled external functions and files, syntax summaries, and references t or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that th command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes---binar hexadecimal---are explicitly indicated.

Part 1
DECthreads Overview and Programming Guidelines

Part 1 contains chapters that provide an overview and concepts of DECthreads as well as defining programming disciplir guidelines for writing a multithreaded program.

Chapter 1
Introducing DECthreads for Multithreaded Programming

This chapter introduces the concepts of threads and multithreaded programming. It describes four functional models that basis for constructing multithreaded applications. The concepts and techniques introduced here are described in more d Chapter 2 and in this guide's platform-specific appendixes.

This chapter's last section introduces the components of the DECthreads package, in particular the **pthread** and **tis** inte how those components support building multithreaded applications and thread-safe libraries.

1.1 Advantages of Using Threads

Multithreaded programming means organizing and coding a program so that instances of its routines, called threads, c concurrently in the same process. You use threads to improve a program's performance---that is, its throughput, comput speed, responsiveness, or some combination.

Using threads can improve a program's performance on uniprocessor systems by permitting the overlap of input, output, slow operations with computational operations. Threads are useful in driving slow devices such as disks, networks, termi printers. A multithreaded program can perform other useful work while waiting for the device to produce its next event, su completion of a disk transfer or the receipt of a packet from the network.

Using threads can also be advantageous when constructing an application's user interface. Consider the typical arrange window system. Each time the user invokes an action (for example, by clicking on a mouse button), the program can use thread to implement the action. If the user invokes multiple actions, multiple threads can perform the actions in parallel.

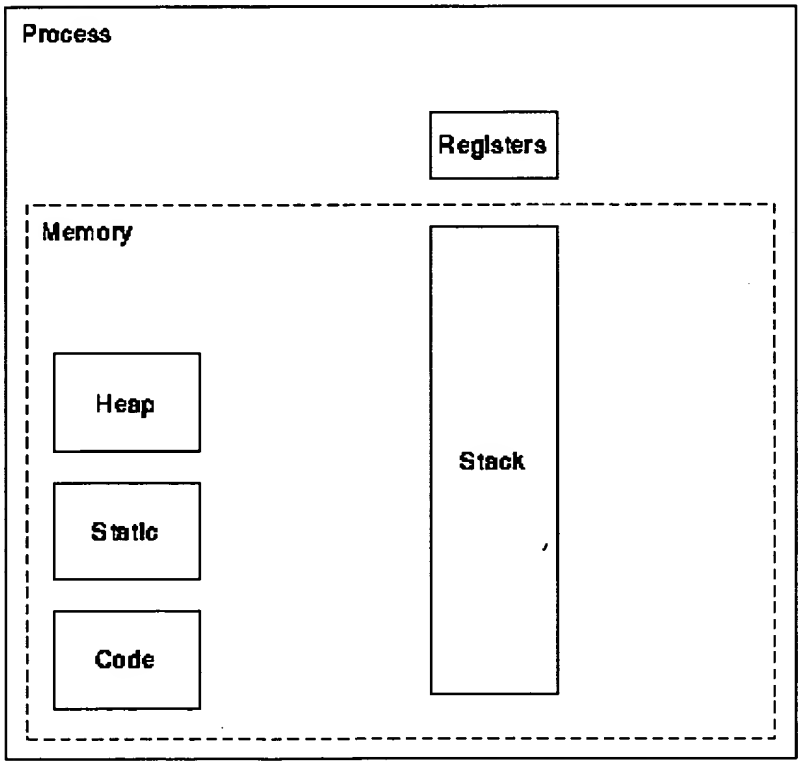
Using threads is especially advantageous when building a distributed system. These systems frequently contain a share server, where the server services requests from multiple clients. Using multiple threads allows the server to handle client in parallel, instead of artificially serializing them or creating (at great expense) one server process per client.

A program with multiple threads can be especially suited to run on a multiprocessor system, where threads run concurre separate processors. Threads created using the DECthreads library are capable of utilizing multiprocessors, if the target supports parallelism within a process. Compaq's DIGITAL UNIX platforms and OpenVMS Alpha platforms support paralle OpenVMS VAX platform does not support parallelism.

1.2 Overview of Threads

A **thread** is a single, sequential flow of control within a process. Within each thread there is a single point of execution. M traditional programs execute as a process with a single thread. Figure 1-1 and Figure 1-2 show the differences between threaded process and a multithreaded process.

Figure 1-1 Single-Threaded Process



ZK-3913A-GE

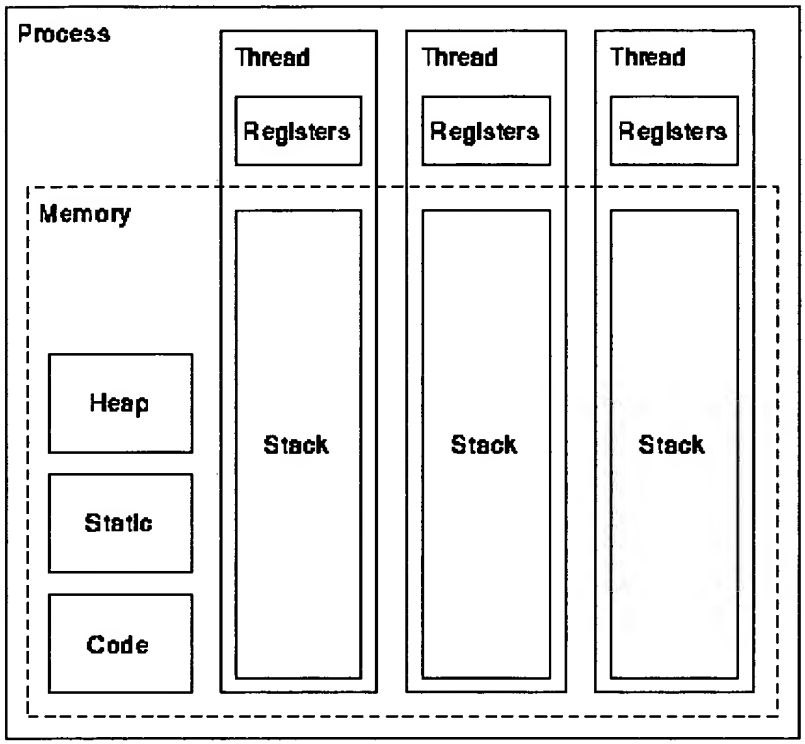
In Figure 1-2, notice that multiple threads share heap storage, static storage, and code but that each thread has its own r and stack.

Using DECthreads, Compaq's multithreading run-time library, a programmer can create several threads within a process process's threads execute concurrently. Within a multithreaded program there are at any time multiple points of executio

Threads execute within (and share) a single address space; therefore, a process's threads can read and write the same locations. When the threads access the same memory locations, your program must use synchronization elements, such mutexes and condition variables, to ensure that the shared memory is accessed correctly. DECthreads provides routines

you to use these and other synchronization objects. Section 2.4 describes the synchronization objects that DECthreads (well as the operations your program can perform on them.

Figure 1-2 Multithreaded Process



ZK-3914A-GE

1.3 Thread Execution

You should design and code a multithreaded program with the assumption that its threads execute *simultaneously*. That program cannot make assumptions about the relative start or finish times of its threads or the sequence in which they ex These are governed by the DECthreads thread scheduler, part of the run-time environment that DECthreads establishes program begins running. Nevertheless, your program can influence how DECthreads schedules its threads, by setting ea scheduling policy and scheduling priority. (Section 2.3.6 describes how thread scheduling works.)

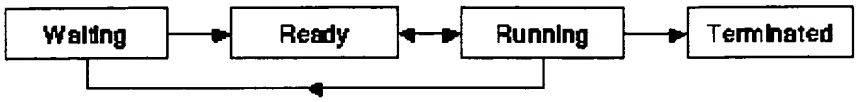
Each thread has its own thread identifier, which distinguishes it from all other threads in the process. In addition to the th scheduling policy and scheduling priority, each thread is associated with any thread-specific instances of thread-commor objects and with thread-specific system resources to support a flow of control.

A thread changes its state over the course of its execution. A thread is in one of the following states:

- *Waiting*---The thread is not eligible to execute, because it is synchronizing with another thread or with an external as I/O.
- *Ready*---The thread is eligible to be executed by a processor.
- *Running*---The thread is currently being executed by a processor.
- *Terminated*---The thread has completed all of its work or has been canceled.

Figure 1-3 shows the transitions between states for a typical thread implementation.

Figure 1-3 Thread State Transitions



ZK-3786A-GE

Note

Building your multithreaded program must produce executable code that is *reentrant*. Therefore, be sure that your compiler generates reentrant code before you design or code your multithreaded program. By default, Compaq's C, C++, Ada, Pascal, and BLISS compilers generate reentrant code.

If you cannot build your program so that its executable code is reentrant, it might be impossible to keep the program's threads from interfering with each other. See Section 3.9.1 for more information about thread-reentrant libraries.

In general, when using threads, be aware of language-based programming practices that are inherently not thread safe. ("Thread safety" is explained in Section 3.9.2.) You must address these factors when writing multithreaded applications and thread-safe libraries. For example, Fortran language routines typically rely heavily upon static storage, which can prevent those routines from being thread safe.

1.4 Functional Models for Multithreaded Programming

The following sections describe four functional models of processing information that are especially well suited for impler multithreaded programs:

- Boss/worker model
- Work crew model
- Pipelining model
- Combination of models

1.4.1 Boss/Worker Model

In a *boss/worker model*, one thread functions as the "boss" because it assigns tasks for "worker" threads to perform. Eac performs a distinct task until it has finished, at which point it notifies the boss that it is ready to receive another task. Alter the boss polls workers periodically to see whether any is ready to receive another task.

A variation of the boss/worker model is the *work queue model*. The boss places tasks in a queue, and workers check the take tasks to perform.

An example of the work queue model in an office environment is a secretarial typing pool. The office manager boss puts to be typed in a basket, and worker typists take documents from the basket to work on.

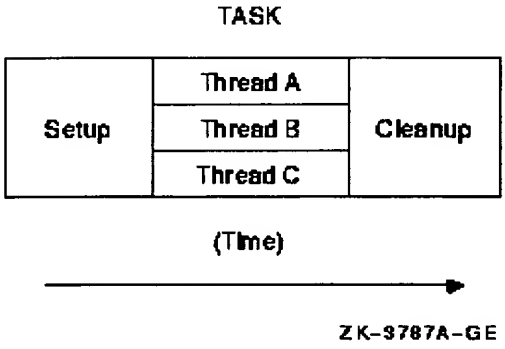
1.4.2 Work Crew Model

In the *work crew model*, multiple threads work together on a single task. The task is divided into pieces that are performe parallel, and each thread performs one piece.

An example of a work crew is a group of people cleaning a building. Each person cleans certain rooms or performs certa work (washing floors, polishing furniture, and so forth), and each works independently.

In a multithreaded program that reflects the work crew model, each thread executes a task that can be performed in pare 1-4 shows a task performed by three threads in a work crew model.

Figure 1-4 Work Crew Model of Thread Operation



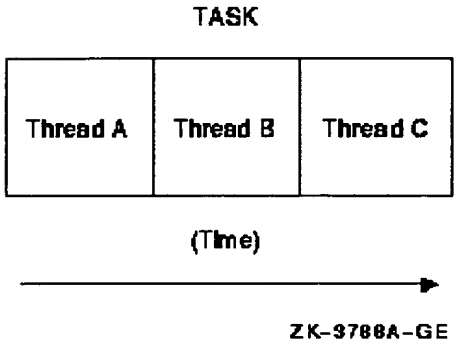
1.4.3 Pipelining Model

In the *pipelining model*, a task is divided into steps. The steps must be performed in sequence to produce a single instan desired result, and the work done in each step (except for the first and last) is based on the previous step and is a prerec the work in the next step. However, the goal is to produce multiple instances of the desired result, and the steps are desi operate in parallel: while one step is performed on one instance of the result, the preceding step can be performed on th instance of the result.

An example of the pipelining model is an automobile assembly line. Each step or stage in the assembly line is continuall receiving the product of the previous stage's work, performing its assigned work, and passing the product along to the ne

In a multithreaded program that reflects the pipelining model, each thread executes a step in the task. Figure 1-5 shows performed by three threads in a pipelining model.

Figure 1-5 Pipelining Model of Thread Operation



1.4.4 Combination of Functional Models

If the task that your program performs is complex, you might find it appropriate to organize it as a combination of the func models previously described. For example, a program could follow the pipelining model, but with one or more steps perfc set of threads that follow a work crew model. In addition, threads could be assigned to a work crew by taking a task from queue and deciding (based on the task characteristics) which threads are needed for the work crew.

1.5 Potential Issues for Multithreaded Programs

When you design and code a multithreaded program, you must accommodate or eliminate, as appropriate, each of the fc issues:

- *Program complexity* is the most significant issue to consider in any multithreaded programming effort. Although u:

can simplify the coding and designing of a program, a certain level of expertise is required to be sure that the des synchronization and interplay among threads is appropriate and correctly specified. This level of expertise is high required to design most single-threaded programs.

- *Dependence upon other nonreentrant software* means that your multithreaded program calls a routine or library tl equipped to deal with threads. Given this dependence, your program must use the DECthreads global lock to pre conflicts with other threads that use the same nonreentrant routine or library. Section 3.9 presents multithreaded programming techniques for managing dependencies upon other nonreentrant software.
- Due to programming errors, *race conditions* in the program's behavior can cause unpredictable and erroneous pr behavior. Similarly, *deadlocks* can cause two or more threads to be blocked from executing indefinitely. Section 3 discusses race conditions in more detail, and Section 3.6.3 discusses deadlocks.
- *Priority inversion* prevents high-priority threads from executing when interdependencies exist among three or mor different priorities. Section 3.5.2 discusses techniques for avoiding priority inversion.

Next | Contents | Index

Upgrading Privileged-Code Applications on OpenVMS Alpha a OpenVMS I64 Systems

Previous | Contents | Index

Chapter 6
Kernel Threads Process Structure

This chapter describes the components that make up a kernel threads process. This chapter contains the following secti

- Section 6.1 describes the process control block (PCB) and the process header (PHD).
- Section 6.2 describes the kernel thread block (KTB).
- Section 6.3 describes the process identifier (PID).
- Section 6.4 describes the process status bits.

For more information about kernel threads features, see the OpenVMS Alpha Version 7.0 Bookreader version of the *Ope Programming Concepts Manual.*

6.1 Process Control Blocks (PCBs) and Process Headers (PHDs)

Two primary data structures exist in the OpenVMS executive that describe the context of a process:

- Software process control block (PCB)
- Process header (PHD)

The PCB contains fields that identify the process to the system. The PCB comprises contexts that pertain to quotas and scheduling state, privileges, AST queues, and identifiers. In general, any information that must be resident at all times is Therefore, the PCB is allocated from nonpaged pool.

The PHD contains fields that pertain to a process's virtual address space. The PHD consists of the working set list, and t section table. The PHD also contains the hardware process control block (HWPCB), and a floating point register save ar HWPCB contains the hardware execution context of the process. The PHD is allocated as part of a balance set slot, and outswapped.

6.1.1 Effect of a Multithreaded Process on the PCB and PHD

With multiple execution contexts within the same process, the multiple threads of execution all share the same address s have some independent software and hardware context. This change to a multithreaded process impacts the PCB and F structures and any code that references them.

Before the implementation of kernel threads, the PCB contained much context that was per process. With the introductio

threads of execution, much context becomes per thread. To accommodate per-thread context, a new data structure---the thread block (KTB)--- is created, with the per-thread context removed from the PCB. However, the PCB continues to con common to all threads, such as quotas and limits. The new per-kernel thread structure contains the scheduling state, pri the AST queues.

The PHD contains the HWPCB, which gives a process its single execution context. The HWPCB remains in the PHD; thi is used by a process when it is first created. This execution context is also called the initial thread. A single threaded pro only this one execution context. Since all threads in a process share the same address space, the PHD continues to des entire virtual memeory layout of the process.

A new structure, the floating-point registers and execution data (FRED) block, contains the hardware context for newly cr kernel threads.

6.2 Kernel Thread Blocks (KTBs)

The kernel thread block (KTB) is a new per-kernel thread data structure. The KTB contains all per-thread context moved PCB. The KTB is the basic unit of scheduling, a role previously performed by the PCB, and is the data structure placed ir scheduling state queues. Since the KTB is the logical extension of the PCB, the SCHED spinlock synchronizes access tc and the PCB.

Typically, the number of KTBs a multithreaded process has, matches the number of CPUs on the system. Actually, the n KTBs is limited by the value of the system parameter MULTITHREAD. If MULTITHREAD is zero, the OpenVMS kernel s disabled. With kernel threads disabled, user-level threading is still possible with DECthreads. The environment is identica OpenVMS environment prior to this release that implements kernel threads. If MULTITHREAD is nonzero, it represents t maximum number of execution contexts or kernel threads that a process can own, including the initial one.

In reality the KTB is not an independent structure from the PCB. Both the PCB and KTB are defined as sparse structures of the PCB that move to the KTB retain their original PCB offsets in the KTB. In the PCB, these fields are unused. In effe structures are overlaid, the result is the PCB as it currently exists with new fields appended at the end. The PCB and KTI initial thread occupy the same block of nonpaged pool; therefore, the KTB address for the initial thread is the same as fo

6.2.1 KTB Vector

When a process becomes multithreaded, a vector similar to the PCB vector is created in pool. This vector contains the lis addresses for the kernel thread blocks in use by the process. The KTB vector entries are reused as kernel threads are cr deleted. An unused entry contains a zero. The vector entry number is used as a kernel thread ID. The first entry always c address of the KTB for the initial thread, which is by definition kernel thread ID zero. The kernel thread ID is used to build PIDs for the individual kernel threads. Section 6.3.1 describes PID changes for kernel threads.

To implement these changes, the following four new fields have been added to the PCB:

- PCB$L_KTBVEC
- PCB$L_INITIAL_KTB
- PCB$L_KT_COUNT
- PCB$L_KT_HIGH

The PCB$L_INITIAL_KTB field actually overlays the new KTB$L_PCB field. For a single threaded process, PCB$L_KTB initialized to contain the address of PCB$L_INITIAL_KTB. The PCB$L_INITIAL_KTB always contains the address of the thread's KTB. As a process transitions from being single threaded to multithreaded and back, PCB$L_KTBVEC is updat to either the KTB vector in pool or PCB$L_INITIAL_KTB.

The PCB$L_KT_COUNT field counts the valid entries in the KTB vector. The PCB$L_KT_HIGH field gives the highest v number in use.

6.2.2 Floating-Point Registers and Execution Data Blocks (FREDs)

To allow for multiple execution contexts, not only are additional KTBs required to maintain the software context, but addit HWPCBs must be created to maintain the hardware context. Each HWPCB has allocated with it a block of 256 bytes for the contents of the floating-point registers across context switches. Another 128 bytes is allocated for per-kernel thread c Presently, only a clone of the PHD$L_FLAGS2 field is defined.

The combined structure that contains the HWPCB, floating-point register save area, and per-kernel thread data is called

point registers and execution data (FRED) block. It is 512 bytes in length. These structures reside in the process's balan This allows the FREDs to be outswapped with the process header. On the first page allocated for FRED blocks, the first are reserved for the inner-mode semaphore.

6.2.3 Kernel Threads Region

Much process context resides in P1 space, taking the form of data cells and the process stacks. Some of these data cell per-kernel thread, as do the stacks. By calling the appropriate system service, a kernel thread region in P1 space is initia contain the per-kernel thread data cells and stacks. The region begins at the boundary between P0 and P1 space at add 40000000x, and it grows toward higher addresses and the initial thread's user stack. The region is divided into per-kerne areas. Each area contains pages for data cells and the four stacks.

6.2.4 Per-Kernel Thread Stacks

A process is created with four stacks; each access mode has one stack. All four of these stacks are located in P1 space. are either fixed, determined by a SYSGEN parameter, or expandable. The parameter KSTACKPAGES controls the size kernel stack. This parameter continues to control all kernel stack sizes, including those created for new execution contex executive stack is a fixed size of two pages; with kernel threads implementation, the executive stack for new execution c continues to be two pages in size. The supervisor stack is a fixed size of four pages; with kernel threads implementation, supervisor stack for new execution contexts is reduced to two pages in size.

For the user stack, a more complex situation exists. OpenVMS allocates P1 space from high to lower addresses. The us placed after the lowest P1 space address allocated. This allows the user stack to expand on demand toward P0 space. V introduction of multiple sets of stacks, the locations of these stacks impose a limit on the size of each area in which they With the implementation of kernel threads, the user stack is no longer boundless. The initial user stack remains semibou still grows toward P0 space, but the limit is the per-kernel thread region instead of P0 space.

Previous	Next	Contents	Index

Exhibit J to

Petition for Review by Technology Center SPRE

Apple Computer Documentation for Threads

Multithreading Programming Topics

2005-03-03

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Introduction to Multithreading Programming Topics

Multithreading is a technique generally used to improve performance and enhance the perceived responsiveness of applications. Especially on computers with more than one processor, multithreading can allow a program to execute multiple pieces of code simultaneously. Of course, multithreading also has its dangers. Protecting critical data structures and synchronizing the efforts of multiple threads requires careful thought and consideration.

This document introduces you to the basic concepts associated with creating a multithreaded application. It also provides examples and guidance for how to use the multithreading technologies in Mac OS X.

Important: The threading and locking classes described in this document are only available for applications that use C-based languages. If you are interested in multithreading for Java applications, see the Java documentation on Sun's website.

Organization of This Document

The various issues involved in building a multithreaded application are covered in the following articles:

- "Threads" (page 11) describes what threads are and why they are useful.
- "Thread Packages" (page 15) describes the advantages and disadvantages of the different threads packages in Mac OS X.
- "Synchronization and Locking" (page 19) describes the technologies available for synchronizing and protecting critical regions of code.
- "Thread Communication" (page 23) discusses the technologies available for communicating between threads.
- "Thread Safety Guidelines" (page 27) offers guidance on how to avoid many of the problems that arise with multiple threads of execution.
- "Cocoa Thread Safety" (page 31) describes safety issues to be aware of when using Cocoa objects in threads.
- "Core Foundation Thread Safety" (page 37) describes safety issues to be aware of when using Core Foundation objects in threads.

- "Creating Threads in Cocoa" (page 39) demonstrates how to create a new thread in a Cocoa application.
- "Creating Threads in Carbon" (page 43) shows how to create a new thread in a Carbon application.
- "Using Locks in Cocoa" (page 47) demonstrates the use of Cocoa locking techniques.
- "Using POSIX Thread Locks" (page 53) demonstrates the use of pthread mutexes and conditions.
- "Communicating With Mach Ports" (page 55) demonstrates the use of Mach ports to communicate between Carbon and Cocoa threads.
- "Communicating With Distributed Objects" (page 61) demonstrates the use of Distributed Objects to communicate between Cocoa threads.

Threads

In Mac OS X, each process comprises one or more threads. A thread is a stream of execution that runs code for the process. Multiple threads may execute the same code, but they do so independently of other threads. Each thread has its own execution stack and is capable of independent input and output. However, all threads share the same virtual memory address space and have the same access rights as the parent process.

Threads let your program perform multiple tasks in parallel. For example, you can use threads to perform several, lengthy calculations while your user interface continues to respond to user commands. You could also use threads to divide a large job into several smaller jobs.

On multiprocessor systems, each processor can execute the code for a different thread. This parallel execution can lead to tremendous performance gains for your program and for the system as a whole.

A process has only one thread initially. That thread can spawn additional threads using any of the available threading APIs described in "Thread Packages" (page 15). The threading API you choose is usually driven by the programming environment you are using (Carbon, Cocoa, Darwin, and so on) and your performance needs.

Note: For a historical look at the threading architecture of Mac OS, and for additional background information on threads, see Technical Note TN2028, "Threading Architectures".

Thread Terminology Issues

If you are familiar with Carbon's Multiprocessor Services API, then you are probably used to seeing the term "task" to represent a separate thread of execution. The use of the term "task" in the Multiprocessor Services API was intended to distinguish from the terminology used by the Carbon Thread Manager API. The term "task" may also be familiar to developers who work on UNIX systems, where the term indicates a running process.

In practical terms, a Multiprocessor Services task is equivalent to a preemptively scheduled thread in Mac OS X. In fact, with one exception, all threads in Mac OS X are scheduled preemptively. Only the Carbon Thread Manager uses a cooperative scheme for scheduling threads. This scheme ensures that applications written for earlier versions of Mac OS will not break when ported directly to Mac OS X.

Exhibit K to

Petition for Review by Technology Center SPRE

Comparative Discussion of Several Vendors' Implementations of Threads

Threads on HP-UX, Solaris, and NT

by
David Graves, HP

This document provides a conceptual mapping of thread function calls between four implementations: HP-UX threads (based on POSIX and X/Open), Solaris threads, POSIX threads on Solaris, and NT proprietary threads.

Overview of implementations

POSIX threads are more portable, establish characteristics for each thread according to c attribute objects, implement thread cancellation, enforce scheduling algorithms, and allow up handlers for fork calls.

Solaris threads can be suspended and continued, implement optimized reader/writer locki increase the concurrency, and implement daemon threads, for whose demise the process wait.

X/Open threads extend the POSIX standard to include many of the capabilities of Solaris such as suspend, continue, and optimized reader/writer locking.

The POSIX implementations on HP-UX and Solaris are very similar. However, HP-UX *pth* include the X/Open threads calls (frequently similar to proprietary Solaris threads calls). A pthreads include non-portable functions (with names ending in `_np`) which sometimes re proprietary Solaris or NT threads calls. Thus, HP-UX pthreads are nearly a superset of PC threads, Solaris threads, and NT threads.

Synchronization

Since threads run concurrently and share resources, synchronization mechanisms are rec provide mutually exclusive access to shared data. The four mechanisms defined in threac are: mutexes, condition variables, reader/writer locking (frequent-read occasional-write), a semaphores. Each are covered in the tables below.

Solaris threads and the HP-UX threads implement all four mechanisms. The Solaris POSI implements all but reader/writer locking. NT implements all four, but there is not a direct m UNIX® functions. For example, multiple reader single writer locking is a feature of IStrean which is not the same as Reader/Writer locks in POSIX threads.

Thread call implementations

These notes correspond to the table that follows.

- Functions marked with an asterisk (*) were developed by X/Open; they are implem

HP-UX version of pthreads, but not in the Solaris version.

- Functions marked with a tilde (~) provide functionality similar to, but not the same functions.
- HP-UX function names ending in _np are **N**ot **P**ortable to other UNIX platforms. For the HP-UX `pthread_resume_np` exactly matches NT's `ResumeThread` (incrementing/decrementing suspension count), while the HP-UX `pthread_cont` matches Solaris' `thr_continue` function.

Creation

POSIX and X/Open	Solaris	NT
pthread_create	thr_create	CreateThread
---	thr_create(daemon)	CreateRemoteThrea
pthread_create(start_func)	thr_create(start)	ThreadProc
pthread_attr_init	---	---
pthread_attr_destroy	---	---
pthread_attr_getdetachstate	---	---
pthread_attr_getguardsize*	---	---
pthread_attr_getinheritsched	---	---
pthread_attr_getschedparam	---	---
pthread_attr_getschedpolicy	---	---
pthread_attr_getstackaddr	---	---
pthread_attr_getstacksize	---	---
pthread_attr_getscope	---	---
pthread_attr_setdetachstate	---	---
pthread_attr_setguardsize*	---	---
pthread_attr_setinheritsched	---	---
pthread_attr_setschedparam	---	---
pthread_attr_setschedpolicy	---	---
pthread_attr_setstackaddr	---	---
pthread_attr_setstacksize	---	---
---	thr_min_stack	---
pthread_default_stacksize_np	---	---
pthread_attr_setscope	---	---
pthread_attr_getprocessor_np	---	---
pthread_attr_setprocessor_np	---	SetThreadIdealProce

Exit

POSIX and X/Open	Solaris	NT
pthread_exit	thr_exit	ExitThread
pthread_join	thr_join	GetExitCodeThread-
pthread_detach	---	---

Printable version